THE UNIVERSITY *of* EDINBURGH

Towards the Development of a Reliable Reconfigurable Real-time Operating System on FPGAs

Chuan Hong

THE UNIVERSITY *of* EDINBURGH

A thesis submitted for the Degree of Doctor of Philosophy

The University of Edinburgh

June 2013

Declaration

I hereby declare that this thesis was composed and originated entirely by myself; work contained herein is my own except where explicitly stated in the text, and it has not been submitted for any other degree or professional qualification.

Chuan Hong

June 2013

Edinburgh, U.K.

Acknowledgements

First of all, I would like to express my sincere appreciation to my first supervisor Dr. Khaled Benkrid, who has given me consistent help and unceasing guidance, with his profound knowledge, kindness and patience extended to me.

I also would like to give my gratitude to my supervisor Dr. Alister Hamilton, for his invaluable assistance in completing this thesis, and all the efforts paid in my last year of study. I am also grateful and indebted to Professor Tughrul Arslan and Dr. Ahmet T. Erdogan, who have given me important assistance in in my academic procedures.

In particular, I would like to thank my senior colleague Dr. Xabier Iturbe, who originally conducted this project and gave me precious suggestions throughout the period of my study. In addition, I record my acknowledgements to Ali Ebrahim, Dr. Hana Mohammad Hussain, Dr. Mohammad Nazrin Isa, and Mr Len Shaw, who have made invaluable contributions to this thesis. The same appreciation goes to my colleagues Mr. Hopsen, Mr. Ding, Mr. Zhang, and Mr. Zhou for their kind encouragement and enlightenment.

I also take this opportunity to record my deepest gratitude and best wishes to my parents, who have made everything possible during my whole life.

Finally, I express my gratitude to all of the SLIG group members, who directly or indirectly lent their helping hands on this trip.

Abstract

In the last two decades, Field Programmable Gate Arrays (FPGAs) have been rapidly developed from simple “glue-logic” to a powerful platform capable of implementing a System on Chip (SoC). Modern FPGAs achieve not only the high performance compared with General Purpose Processors (GPPs), thanks to hardware parallelism and dedication, but also better programming flexibility, in comparison to Application Specific Integrated Circuits (ASICs). Moreover, the hardware programming flexibility of FPGAs is further harnessed for both performance and manipulability, which makes Dynamic Partial Reconfiguration (DPR) possible. DPR allows a part or parts of a circuit to be reconfigured at run-time, without interrupting the rest of the chip’s operation. As a result, hardware resources can be more efficiently exploited since the chip resources can be reused by swapping in or out hardware tasks to or from the chip in a time-multiplexed fashion. In addition, DPR improves fault tolerance against transient errors and permanent damage, such as Single Event Upsets (SEUs) can be mitigated by reconfiguring the FPGA to avoid error accumulation. Furthermore, power and heat can be reduced by removing finished or idle tasks from the chip. For all these reasons above, DPR has significantly promoted Reconfigurable Computing (RC) and has become a very hot topic. However, since hardware integration is increasing at an exponential rate, and applications are becoming more complex with the growth of user demands, high-level application design and low-level hardware implementation are increasingly separated and layered. As a consequence, users can obtain little advantage from DPR without the support of system-level middleware.

To bridge the gap between the high-level application and the low-level hardware implementation, this thesis presents the important contributions towards a Reliable, Reconfigurable and Real-Time Operating System (R3TOS), which facilitates the user exploitation of DPR from the application level, by managing the complex hardware in the background. In R3TOS, hardware tasks behave just like software tasks, which can be created, scheduled, and mapped to different computing resources on the fly. The novel contributions of this work are: 1) a novel implementation of an

efficient task scheduler and allocator; 2) implementation of a novel real-time scheduling algorithm (FAEDF) and two efficacious allocating algorithms (EAC and EVC), which schedule tasks in real-time and circumvent emerging faults while maintaining more compact empty areas. 3) Design and implementation of a fault-tolerant microprocessor by harnessing the existing FPGA resources, such as Error Correction Code (ECC) and configuration primitives. 4) A novel symmetric multiprocessing (SMP)-based architectures that supports shared memory programing interface. 5) Two demonstrations of the integrated system, including a) the K-Nearest Neighbour classifier, which is a non-parametric classification algorithm widely used in various fields of data mining; and b) pairwise sequence alignment, namely the Smith Waterman algorithm, used for identifying similarities between two biological sequences.

R3TOS gives considerably higher flexibility to support scalable multi-user, multi-tasking applications, whereby resources can be dynamically managed in respect of user requirements and hardware availability. Benefiting from this, not only the hardware resources can be more efficiently used, but also the system performance can be significantly increased. Results show that the scheduling and allocating efficiencies have been improved up to 2x, and the overall system performance is further improved by ~2.5x. Future work includes the development of Network on Chip (NoC), which is expected to further increase the communication throughput; as well as the standardization and automation of our system design, which will be carried out in line with the enablement of other high-level synthesis tools, to allow application developers to benefit from the system in a more efficient manner.

Related Publications

Journals

1. **Efficient On-Chip Task Scheduler and Allocator for Reconfigurable Operating Systems**
   Chuan Hong, Khaled Benkrid, Xabier Iturbe, Ali Ebrahim, and Tughrul Arslan
   IEEE Embedded Systems Letters, vol. 3, issue. 3, pp. 85–88, 2011.

2. **Efficient Run-time System Support for High Performance Reliable Reconfigurable Systems**
   Chuan Hong, Khaled Benkrid, Xabier Iturbe and H. Hussain
   Journal of Computational Intelligence and Electronic Systems (JCIES), 2013, (Accepted for publication)

3. **A Run-time Reconfigurable System for Adaptive High Performance Efficient Computing**
   Chuan Hong, Khaled Benkrid, Nazrin Isa and Xabier Iturbe
   ACM SIGARCH Computer Architecture News, 2013, (Accepted for publication)

4. **R3TOS: A Novel Reliable Reconfigurable Real-Time Operating System for Highly Adaptive, Efficient and Dependable Computing on FPGAs**
   Xabier Iturbe, Khaled Benkrid, **Chuan Hong**, Ali Ebrahim, Raul Torrego, Imanol Martinez, Tughrul Arslan and J. Perez
   IEEE Transactions on Computers, Special Issue on "Adaptive Hardware and Systems", 2013 (Accepted for publication)

5. **Runtime scheduling, allocation and execution of real-time hardware tasks onto Xilinx FPGAs subject to fault occurrence**
   Xabier Iturbe, Khaled Benkrid, **Chuan Hong**, Ali. Ebrahim, Tughrul Arslan and Imanol Martinez
   International Journal of Reconfigurable Computing, 2013 (Accepted for publication)

6. **Microkernel Architecture and Hardware Abstraction Layer of a Reliable Reconfigurable Real-Time Operating System (R3TOS)**
   Xabier Iturbe, Khaled Benkrid, **Chuan Hong**, Ali Ebrahim, Raul Torrego and Tughrul Arslan
   ACM Transactions on Reconfigurable Technology and Systems (Submitted for publication)

Conferences

1. **Design and Implementation of Fault-tolerant Soft Processors on FPGAs**
   Chuan Hong, Khaled Benkrid, Xabier Iturbe and Ali Ebrahim
   International Conference on Field-Programmable Logic and Applications (FPL), 2012, pp.683–686,2012,Oslo, Norway

2. **An FPGA Task Allocator with Preliminary First-Fit 2D Packing Algorithms**
   Chuan Hong, Khaled Benkrid, Xabier Iturbe, Ahmet T. Erdogan, and Tughrul Arslan
   NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp. 264–270, 2011, California, USA.

3. **Virtual Shared Memory Architecture for Inter-Task Communication in Partial Reconfigurable Systems**
   Chuan Hong, Khaled Benkrid, Ali Ebrahim and Xabier Iturbe
   International Conference on Microelectronics (ICM), pp. 1–4, 2012, Algiers, Algeria.

4. **An Adaptive FPGA Implementation of Multi-Core K-Nearest Neighbour Ensemble Classifier using Dynamic Partial Reconfiguration**
   Hanaa Hussain, Khaled Benkrid, **Chuan Hong**, and Huseyin Seker
   International Conference on Field-Programmable Logic and Applications (FPL), pp.627–630, 2012, Oslo, Norway.

5. **Enabling FPGAs for future deep space exploration missions: Improving fault-tolerance and computation density with R3TOS**
   Xabier Iturbe, Khaled Benkrid, Tughrul Arslan, **Chuan Hong**, Ahmet T. Erdogan and Imanol Martinez

NASA/ESA Conference on Adaptive Hardware and Systems, pp. 264–270, 2011, California, USA.

6. **Empty Resource Compaction Algorithms for Real-Time Hardware Tasks Placement on Partially Reconfigurable FPGAs Subject to Fault Occurrence**
   Xabier Iturbe, Khaled Benkrid, Tughrul Arslan, **Chuan Hong**, and Imanol Martinez
   Conference on Reconfigurable Computing and FPGAs, pp. 27–34, 2011, Cancun, Mexico

7. **A Novel High-Performance Fault-Tolerant ICAP Controller**
   Ali Ebrahim, Khaled Benkrid, Xabier Iturbe, and **Chuan Hong**
   NASA/ESA Conference on Adaptive Hardware and Systems, 2012, Erlangen, Germany. (Accepted for publication)
8. **Snake: An Efficient Strategy for the Reuse of Circuitry and Partial Computation Results in High-Performance Reconfigurable Computing**
   Xabier Iturbe, Khaled Benkrid, Ali. Ebrahim, **Chuan Hong**, Tughrul Arslan , and Imanol Martinez
   Conference on Reconfigurable Computing and FPGAs, pp. 182–189, 2011, Cancun, Mexico
9. **Multiple-Clone Configuration of Relocatable Partial Bitstreams in Xilinx Virtex FPGAs**
   Ali Ebrahim, Khaled Benkrid, Xabier Iturbe, **Chuan Hong**
   NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2013. (Accepted for publication)
10. **High Performance Gapped BLAST with the Two-hit Method Implementation on FPGA**
    M.Nazrin M.Isa, Khaled Benkrid, **Chuan Hong**, Thomas Clayton
    Highly Efficient Accelerators and Reconfigurable Technologies (HEART), 2013, (Accepted for publication)

Publications Downloading Link:

http://www.see.ed.ac.uk/~s1048456/

Contents

List of Figures

List of Tables

List of Equations

List of Algorithms

List of Abbreviations

AM	Area-Matrix
AM-DL	AM-Down-Left
AM-Down-Right	AM-DR
AM-UL	AM-Up-Left
AM-UR	AM-Up-Right
API	Application Programing Interface
ASIC	Application Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
BF	Best-Fit
BM	Bus Macro
BRAM	Block RAM
CLB	Configurable Logic Block
CRC	Cyclic Redundancy Check
DM	Deadline Monotonic
DPR	Dynamic Partial Reconfiguration
DRT	Data Reallocation Task
EAC	Empty Area Compaction
EAD	Empty Area Descriptor
ECC	Error Correction Code
ECC-BRAM	ECC-protected BRAM
EDF	Earliest Deadline First
EDK	Embedded Development Kit
EM	Electro Migration
EPA	ECC Processor Adaptor
ESD	Electrostatic Discharge
EVC	Empty Volume Compaction
FAEDF	Finishing Aware Earliest Deadline First
FAR	Frame Address Register
FCCR	Fault Containment Computation Region
FCU	Fault Containment Unit

FDRI	Frame Data Register Input
FDRO	Frame Data Register Output
FF	First-Fit
FIT	Failures In Time
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
GUI	Graphic User Interface
HBC	High Bandwidth Communication
HCI	Hot Carrier Injection
HDL	Hardware Description Language
HLL	High-Level programming Languages
HPI	Hardware Programing Interface
HWμK	Hardware Microkernel
ICAP	Internal Configuration Access Port
IDB	Input Data Buffer
IDF	Isolation Design Flow
IOB	Input/ Output Block
IVT	Isolation Verification Tool
JTAG	Joint Test Action Group
K-NN	K-Nearest Neighbour
LBC	Low Bandwidth Communication
LUT	Look-Up-Table
MER	Maximum Empty Rectangle
MFW	Multiple Frame Writing
MPI	Message Passing Interface
MTBE	Mean Time Between Errors
MTBF	Mean Time Between Failure
MTTD	Mean Time To Detect
MTTF	Mean-Time-To-Failure
MTTM	Mean Time To Manifest
MTTR	Mean Time To Repair
ODB	Output Data Buffer

PC	Program Counter
PCAP	Processor Configuration Access Port
PCB	Printed Circuit Board
PE	Processing Element
PIP	Programmable Interconnection Points
PR	Partial Reconfigurable
QoS	Quality of Service
R3TOS	Reliable, Reconfigurable and Real-Time Operating System
RC	Reconfigurable Computing
RMB	Reconfigurable Multiple Bus
RNoC	Reconfigurable Network on Chip
ROS	Reconfigurable Operating System
RP	Routing Primitive
RTC	Resource type check
SA	Sequence Alignment
SB	Switch Box
SDR	Software Defined Radio
SEU	Single Event Upset
SIL	Safety Integrity Level
SM	Shape-Matrix
SM-L	SM-Left
SMP	Symmetric MultiProcessing
SM-R	SM-Right
SoC	System on Chip
SRAM	Static RAM
S-W	Smith-Waterman
TCL	Task Control Logic
TDDB	Time-Dependent Dielectric Breakdown
TM	Time Matrix
TMR	Triple Modular Redundancy
VLS	Vertex List Set
XST	Xilinx Synthesis Technology

Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) have been rapidly developed in the last thirty years, offering electronic system developers the high performance of dedicated hardware with the flexible reprogrammability [Kuon2008, Tse2012]. With process technology scale-down, modern FPGAs have achieved significant levels of performance, programming flexibility, power efficiency and convenience [Compton2002, Todman2005, and Tian2010]. FPGAs achieve high speed performance by benefiting from the characteristics of their hardware architecture, such as computing parallelism and circuitry customization, which allows multiple data streams to be processed in parallel, directly through dedicated combinatorial logic gates [Banerjee2006, Sundarajan2010]. This computing parallelism and circuitry customization have considerably increased computing throughput compared with software programmed Von-Neumann architectures, where each instruction has to be fetched, decoded, and executed in a sequential machine [Backus1978]. Moreover, FPGAs give hardware programming flexibility, allowing them to be reconfigured after manufacture. This is achieved by using a configuration memory to dictate all FPGA functionalities. Thus, by changing the content of the configuration memory, the circuit logic can be changed accordingly [Xilinx2009]. Moreover, power efficiency is improved by the dedicated logic circuitry of FPGAs, thanks to their fine grained programming granularity, which allows hardware to be thoroughly customized for a particular application [Sundarajan2010]. Last, but not least, with an

increasing number of FPGA design tools and libraries available, developments of FPGAs have become more convenient and productive, making the time-to-market shorter. To date, FPGAs can be programmed through various design flows, from traditional schematic design, via Hardware Description Language (HDL), to High-Level programming Languages (HLL), allowing the developer to programme FPGAs from either a structuralized viewpoint, or at a more behavioural level, by trading off their productivity with design efficiency [Hutchings1999,Benkrid2000, Thomas2002, Benkrid2004, Benkrid2008]. Figure 1.1 lists four classical computing platforms compared with FPGAs in respect of performance, programmability, productivity, and power efficiency. In general, better performance and power efficiency can be achieved by application-specified hardware, whereas general-purpose software programmed devices have higher programmability and productivity. Among these, the FPGA not only outperforms software programmed devices, for example General Purpose Processors (GPPs) and Application-Specific Instruction-set Processors (ASIPs) in terms of speed, but it also enjoys additional programming flexibility compared with Application-Specific Integrated Circuit (ASICs). In effect, although serving as a middle-level solution, FPGAs are more hardware oriented; that is to say, their speed performance is more prominent than their programmability. For instance, most FPGAs have performance and power efficiency close to ASICs; however, their level of programmability is far lower than that of software programed processors [Benkrid2008]. Therefore, initially, the reconfigurability of FPGAs was only used for design prototyping, through off-line and static full reconfiguration. Making better use of the reconfigurability of FPGAs subsequently became a major research topic [Iturbe2010, Wigley2001, and Mignolet2003].

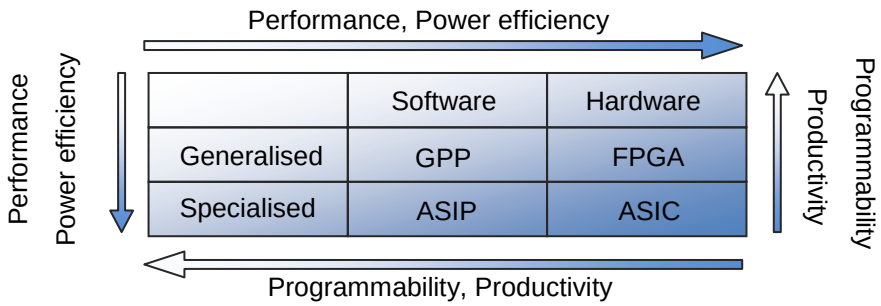


Figure 1.1 FPGA compared with other computing platforms

Dynamic Partial Reconfiguration (DPR) extends programming flexibility to a higher level, and erodes the traditional boundary between software and hardware, giving hardware a software look-and-feel [Brebner1996, Wigley2001, Mignolet2003, and So2007]. DPR allows part of the chip region to be reconfigured without interrupting the operation of the rest of the chip. Benefitting from this, the logic resources can be more efficiently re-used by swapping tasks in/out of the chip in a time-multiplexed fashion on-the-fly [Xilinx2010a]. Therefore, by applying DPR, hardware tasks can be generated, scheduled, and mapped to the two-dimensional hardware resources, which is similar to the software tasks of creating, scheduling and allocating in one-dimensional memory space. By harnessing the configurability of FPGAs, the reconfiguration overhead is further reduced in terms of both time and area, which makes more rapid run-time reconfiguration possible, such as the Xilinx Virtex4 family FPGA, which achieves 32bit bandwidth working at 100MHz, making it potentially capable of configuring a task in microseconds [Xilinx2010b]. DPR brings three main advantages. Firstly, a limited silicon area can be more efficiently shared by swapping tasks over time [Dubach2010]. Secondly, reliability is gained by reconfiguring transient soft-error in the faulty region, or reallocating hardware tasks away from damaged resources [Iturbe2010a]. Finally, power dissipation is reduced since non-running or idle tasks can be swapped out of a chip as and when needed [Iturbe2010b] .

DPR is still not mature enough to be applied widely because of the gap between high-level user applications and low-level hardware details [Coussy2008]. In effect, user software applications have become more complex because of the growth of customer demands, whereas the hardware has become more densely and heterogeneously integrated in recent decades. Therefore, developers are normally either specialised at the application level or in hardware implementation, making the design layered. To allow the application developer to benefit from DPR without knowing the details of hardware, a system level middleware support needs to be provided to bridge this gap. Encouraged by these ideas, various approaches have been proposed to exploit the advantages of DPR. For example, the leading FPGA vender, Xilinx Corp., have contributed a series of innovations of DPR, from

hardware chip support, such as Internal Configuration Access Port (ICAP), via design flow solutions, to software Application Programing Interfaces (APIs), including the PR-PlanAhead, and MicroBlaze ICAP driver [Xilinx2010a, Xilinx2010b]. With such support from industrial providers, academic research has focused more on system level design flows [Horta2002, Blodget2003]. However, most of this research has not gone beyond concepts and simulations, with insufficient realistic application support. And research usually specializes in limited features of DPR, with a lack of system generality and applicability [Brebner1996, Wigley2001, Kalte2005 and Becker2007].

In light of the above, a software-like Reliable Reconfigurable Real-time Operating System (R3TOS) has been developed on FPGAs, in the belief that the widespread use of DPR will not materialize and become generalised until a generic software-like programming model is developed for it [Brebner1996, Wigley2001, Mignolet2003, Walder2005, and So2007]. R3TOS is a generic reconfigurable system that significantly exploits up-to-date DPR technology, and provides a generalised API for high level applications. R3TOS opens a new model of Reconfigurable Computing (RC) where hardware tasks behave just like software tasks, as they can be scheduled, preempted, and flexibly placed at arbitrary positions on the chip, whereby reliability, reconfigurability, and real-time performance are improved [Iturbe2010a, Iturbe2010b and Iturbe2013b].

1.1 Scope and Objectives

R3TOS is designed to maximally exploit the programming flexibility of DPR, and to universalise the standard OS API for modern FPGAs, in order to facilitate various advanced applications. There are three main objectives can be achieved, namely, reliability, reconfigurability and real-time operation.

- **Reliability**

To date, most commercial FPGAs are based on Static RAM (SRAM), which is highly susceptible to radiation-induced faults, such as Single Error Upsets (SEUs)

provoked by high energy particles in space [Xilinx2011a]. The Rosetta Experiment shows that at the 90nm technology node, the Xilinx Spartan-3 FPGA series suffers from an error rate of 111 FIT/Mb in their Configurable Logic Blocks (CLBs), and 222 FIT/Mb in Block RAMs, (BRAMs), where FIT stands for "failures in time per billion hours" [Lesea2005]. To improve system fault tolerance, the R3TOS improves its reliability in regard to four aspects: 1) fault self-containing (see chapter 3), 2) transient error mitigation, 3) permanent damage avoidance, and 4) prolonging device life time.

1) First of all, in the R3TOS, hardware tasks are both physically and operationally independent, keeping faults which occur self-contained within each module itself, without affecting the rest of the system. This is because all hardware tasks in the R3TOS are exclusively area-constrained within a square region, rather than being mixed up or overlapping with other modules. Consequently, an emerging fault in a particular task is also area-isolated from other tasks, which eases the diagnosis and correction of upcoming errors. Moreover, inter-task communication, for example in giving inputs to a task and receiving outputs from a task, is made error proof by adding redundant checking bits. Therefore, errors are self-contained within tasks and checked individually, improving the overall system fault tolerance [Iturbe2009, Iturbe2011d, and Iturbe2013b].

2) Transient errors are soft-errors provoked by a strike from electro-magnetic radiation irons, causing a state change in a flip-flop or a register in memory or a logic gate, such as Single Event Upsets (SEUs). Conventionally SEUs are eliminated by resetting the whole device to initialise all states [Heiner2008]. For instance, if an SEU occurs in the FPGA's configuration memory, the FPGA has to be shut-down and powered-up again to reinitialise all the configuration states, resulting in temporary data loss. DPR gives the potential to read back the configuration data from the configuration memory, as well as to partially reconfigure a particular region at run-time, which can be used to minimise the chance of an SEU spreading without halting the device. Based on this, the R3TOS applies a scrubbing technique which periodically reads the configuration memory, detects errors, and reconfigures error

frames in the first instance. Thereby the Mean-Time-To-Repair (MTTR) is reduced and error infection is minimised [Ebrahim2012, Iturbe2011d and Iturbe2013b].

3) A damaged resource is an unfixable broken piece of hardware in the chip silicon, which is a hard-error that cannot be repaired. The traditionally used Triple Modular Redundancy (TMR) method triplicates the hardware module in order to retain the correctness in cases where one of the three modules fails [Xilinx2001]. However, it is not fully error proof, since the whole triplicated module will fail if more than one module is damaged. In the R3TOS, damaged resources can be avoided by reallocating tasks to another non-damaged region; therefore, the accumulation of damage will not reduce the Mean-Time-To-Failure (MTTF) but only reduce the number of available resources [Hong2011b, Iturbe2011d, and Hong2012a].

4) The device may become worn out once faults or damages can no longer be tolerated. The device's overall lifetime is bottlenecked by the chip's critical region which is most intensively used and thus ages more quickly and is worn out earliest. In regard to this, a method is proposed to prolong device lifetime by distributing the usage intensity around the chip. Here, hardware tasks are not fixed at a particular region, but are swapped in/out over the whole chip, making the whole chips' resources more evenly used. Hence, the device resources age uniformly and the overall chip lifetime is prolonged [Iturbe2011d, Iturbe2013a].

- **Reconfigurability**

DPR gives FPGAs high programming flexibility with less reconfiguration overhead in terms of both speed and area. However, most of the existing approaches do not fully exploit the programming flexibility provided by DPR. First, and most commonly, such approaches use a predefined fixed Partial Reconfigurable (PR) region or regions to accommodate hardware tasks, whereby boundaries have to be fixed with static communication ports such as Bus Macros (BMs) or Proxy Logic. As a result, hardware resources are wasted since the PR regions have to cater for the largest reconfigurable task possible [Iturbe2010a, Hong2012b]. Secondly, hardware tasks are usually pre-synthesized off-line. Therefore, only a limited number of tasks

are available at run-time, and multiple versions of task bitstreams consume large memory resources, resulting in resource dissipation. Moreover, the static nature of PR regions and fixed communication ports makes it more difficult to reconfigure around emerging faults.

The R3TOS allows tasks to be placed at arbitrary positions with no boundary concerns. This is achieved by means of a novel inter-task communication mechanism, which relies on: 1) ICAP-based inter-task communication; and 2) Data Reallocation Tasks (DRTs). The ICAP-based communication harnesses the internal configuration port and uses it to exchange data at the configuration layer, leading to a clean "route-less" reconfiguration region. Due to the limitations of the bandwidth of the configuration port, ICAP-based communication is only used for Low Bandwidth Communication (LBC) tasks, namely those dominated by computation. For example, to allow High Bandwidth Communication (HBC) tasks, that is communication dominated tasks, DRTs are used such as data streaming applications. DRT is a temporarily allocated module used to link two existing hardware tasks, and it can provide the high bandwidth required in intensive communication [Iturbe2011b]. In addition, two optional communication architectures are also presented, namely, the shared memory based architecture and the snake allocation strategy. The shared memory architecture is inspired by the Symmetric Multiprocessing (SMP) paradigm in multiprocessor programming [Hong2012b]. In it, the physically isolated Block RAMs (BRAMs) are synchronized to keep the same content shared by all tasks. The snake allocation strategy, on the other hand, geographically explores the possibility of allowing allocated tasks to reuse the intermediate partial results between different stages of computation [Iturbe2011a]. All of the above communication mechanisms make the inter-task communication in the R3TOS possible without using conventional routings, leading to a clean and free reconfigurable area which tasks can be arbitrarily allocated.

With the assistance of the aforementioned inter-task communication, the R3TOS is able to flexibly place an upcoming task anywhere on the chip. To take full advantage of the high flexibility, 2D-packing algorithms can be used to find optimum locations,

in order to reduce chip fragmentation and retain more compact areas for later tasks [Bazargan2000, Tabero2004]. In particular, two novel placing algorithms are implemented by the author; namely, the Empty Area Compaction (EAC) algorithm and the Empty Volume Compaction (EVC) algorithm, which show better simulation results compared with previous algorithms in terms of both efficacy and feasibility [Iturbe2010c, Iturbe2011c, Hong2011a, Hong2011b, and Iturbe2013a]. Besides this, a new ICAP driver is also developed by the group member Ali Ebrahim [Ebrahim2012], which utilises the compressed bitstream configuration, and achieves better speed performance compared with commercial ICAP drivers such as the Xilinx HwICAP IP core. In addition, the R3TOS also supports on-line task generation, whereby hardware tasks can be customized with optimum parameters taking into consideration both user requirements and currently available resources [Hong2013b].

The high flexibility in reconfiguration allows the R3TOS to use limited hardware resources more efficiently. It also provides higher freedom to allow for high performance computing systems to autonomously adapt to user demands and environmental changes. For instance, the R3TOS supports scalable multi-user, multi-tasking applications whereby resources can be dynamically managed in respect of user requirements and hardware availability.

- **Real-Time Operating**

The R3TOS is a real-time operating system, with relatively high predictability of hardware scheduling and operating. To achieve this, a novel scheduling algorithm is implemented by the author called the Finishing Aware Earliest Deadline First (FAEDF) algorithm, which improves the traditional Earliest Deadline First (EDF) algorithm by delaying the execution of tasks according to the current available hardware resources, resulting in better scheduling efficiency [Iturbe2013a]. In addition, real-time performance is further improved by hardware predictability, which gives relatively predictably operating clock cycles. Last, but not least, the R3TOS software API utilises FreeRTOS (an open source real-time operating system), which is an open standard real-time operating system, to provide a more

user friendly interface, as well as more precise prediction of performance [Inam2011, Iturbe2013b].

- **Case Studies**

In this thesis, the R3TOS is demonstrated in the context of two applications, namely the K-Nearest Neighbour (K-NN) classifier, and Sequence Alignment (SA). K-NN is a non-parametric classification algorithm which is used to compute distances between a query sample and all members of a training set. In our demonstration, each K-NN module can be placed and reallocated arbitrarily on the chip, and therefore the chip's resources are more efficiently exploited and fault tolerance is achieved due to modular reconfigurability [Hong2013a]. In the SA application, the Smith-Waterman algorithm is implemented using pipelined Processing Elements (PEs), whose pipeline-stages can be parameterized according to the current available resources and user requirements [Hong2013b]. The results from the above applications show that the R3TOS achieves high flexibility in task customization, scheduling, and placement, whereby resource efficiency and high performance are improved with minimal time and area overhead.

1.2 Novelty and Contributions

First of all, a novel implementation of an efficient task scheduler and allocator is presented. The task scheduler is responsible for scheduling hardware tasks in real-time efficiently, according to their time constraints; and the task allocator is used to search for an optimal position to place the hardware task.

In line with the task scheduler and allocator, a novel implementation of a real-time scheduling algorithm, called Finishing Aware Earliest Deadline First (FAEDF) algorithm, and two efficacious allocating algorithms, namely Empty Area Compaction (EAC) algorithm and the Empty Volume Compaction (EVC) algorithm, are also presented in this work. Results show that the our scheduling algorithm has outperformed other scheduling algorithms with up to 2x scheduling efficiency, and the allocating algorithm has achieved ~25% improvement on allocating efficiency.

Moreover, the contribution also includes the design and implementation of a novel fault-tolerant microprocessor. Based on Xilinx's PicoBlaze processor core, the fault-tolerant microprocessor achieves a high reliability by harnessing the existing FPGA resource, such as Error Correction Code (EDD) and configuration mechanisms. Results show that the fault-tolerant microprocessor can automatically correct a single bit error, as well as reliably recover from multiple errors.

In addition, a novel symmetric multiprocessing (SMP)-based architectures is developed, which enables the system to be programed in a shared memory-like interface, with low area and time overheads.

Last but not least, the system is demonstrated in the context of two applications: the K-Nearest Neighbour classifier (K-NN) and the pairwise sequence alignment (SA). The K-NN classifier is a non-parametric classification algorithm which has been widely used in a number of data mining applications; and the SA application uses the Smith Waterman algorithm to identify the similarities between two biological sequences. Results show that by using the proposed reconfigurable system, the overall computing performance is improved by ~2.5x.

1.3 Outline of the Thesis

The remainder of the thesis is structured as follow:

Chapter 2 - Work Relevant to Dynamic Partial Reconfiguration

This chapter discusses the basic concepts of RC, and previous related approaches. Concepts, methodologies, and performance are evaluated in order to develop the R3TOS project.

Chapter 3 - R3TOS System Overview

This chapter outlines the R3TOS's basic principles, highlighting its major capabilities and representative features. The chapter presents the basic ideas of how R3TOS bridges the gap between low-level hardware and a high-level OS-like API, and turns electronic reconfigurable advances into system performance.

Chapter 4 – Design of Algorithms for the Scheduling and Allocation of Tasks

This chapter presents three R3TOS algorithms designed for real-time task scheduling and hardware allocation, including one efficient scheduling algorithm called Finishing Aware Earliest Deadline First (FAEDF) and two allocating algorithms; namely, Empty Area Compaction (EAC) algorithm and the Empty Volume Compaction (EVC) algorithm. The algorithms are illustrated in the context of an R3TOS simulated environment and their performance is evaluated and compared with that of other state-of-the-art algorithms.

Chapter 5 - Hardware Implementation

This chapter presents the detailed hardware implementations of the components in the R3TOS. The system consists of four main components: the task scheduler, the task allocator, the ICAP manager, and the host API. The hardware architectures of each component are described, including the program flow and the memory organization. This chapter gives details of the technical support needed for implementing the methodologies described in the previous chapters.

Chapter 6 – Development of R3TOS Applications

This chapter presents two representative applications to demonstrate R3TOS in real practice, and gives further detailed explanations about system operation. The first is the K-Nearest Neighbour classifier, which is a classification algorithm used in data mining fields, while the second is a sequence alignment application in which the Processing Elements (PEs) are scaled and parameterized on-line to benefit from the system's reconfigurability. The results are presented to validate the improvements in system performance brought by the R3TOS.

Chapter 7 – Conclusion and Future Work

This chapter draws the main conclusions of the thesis and points towards future work.

Chapter 2

Work Relevant to Dynamic Partial Reconfiguration

DPR-based technologies have been increasingly developed in the last decade by both industrial companies and researchers. These technologies include a series of supports from the chip vendors; customer-designed tools and methodologies; and the development of reconfigurable operating systems and high-level applications. During this period, industrial FPGA vendors have played a leading role, as they provided physical support for technologies associated with DPR. In addition, FPGA vendors have also integrate features of DPR into their IDE tool suites and commercial APIs, such as the ICAP driver, and DPR synthesising. However, since the hardware resources are becoming more and more dense and heterogeneous, the manipulation of the bitstream at runtime becomes both more risky and more complex. As a consequence, for safety issues, most of the industrial tools have been developed conservatively with a lack of programming flexibility. To further enhance configurability, researchers have proposed a number of customer-defined tools and methods which can effectively solve a number of difficulties in managing complex hardware resources, such as: resource heterogeneity, on-chip routing, and inter-task communication. These tools and methods automate control of the low-level hardware and provide a relatively simple Hardware Programing

Interface (HPI) for upper-level designs such as the development of the ROS. The ROS supports more standardised OS-like user interfaces for high-level applications, giving hardware tasks a software look-and-feel by managing hardware resources in its background. The whole design chain has promoted DPR from being a complicated piece of hardware to an automated and generic user interface with friendly OS-like API, bridging the gap between high-level applications and low-level hardware implementation (see Figure 2.1).

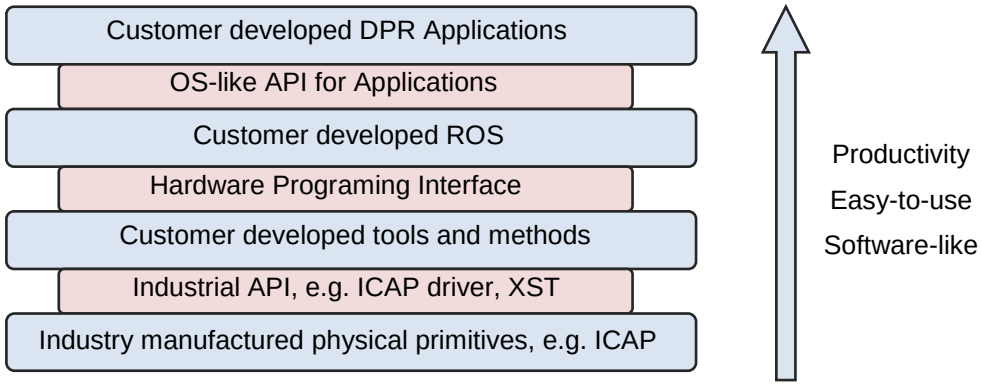


Figure 2.1 DPR Development Chain

This chapter discusses previous approaches relevant to DPR, from low-level industrial supports, via middle-level costumer-designed tools, to high-level ROSs. Concepts, methodologies, and performance are evaluated in order to develop the R3TOS project.

2.1 Dynamic Partial Reconfiguration Supports from Industry

In the last decade, industrial companies have promoted the development of FPGAs in various respects, such as transistor density, heterogeneous hardware integration, and development tools. These developments have significantly improved FPGAs in terms of speed performance, power consumption, and time-to-market. In particular, the techniques associated with DPR have been increasingly promoted by continually updating products in response to technology growth and customer demands. For

instance, the hardware ICAP was developed by Xilinx to enable configuration memory to be accessed from the internal chip logic, giving higher configuration throughput and lower configuration overhead [Xilinx2010b]. To date, FPGA reconfigurability has been dramatically enhanced, and a number of products have been commercialised by FPGA vendors, including DPR-supported devices, design flows, hardware drivers, and software integrated tools [Xilinx2007, Xilinx2010b, and Xilinx2012a, Dye2010, Korf2011, Wigley2001].

2.1.1 Device Support

Modern FPGAs provide the ability to reconfigure chip on-the-fly. A generic FPGA consists of reconfigurable logic cells, reconfigurable input/output cells, reconfigurable routings and specific function blocks (see Figure 2.2).

The reconfigurable logic cells are the basic elements that can be configured to any combinatorial logic, such as logic gates and flip-flops. To build larger system logic, the reconfigurable logic cells are connected together by reconfigurable interconnections, which can be reconfigured to provide arbitrary links between logic cells. The reconfigurable IO cells are used for interfacing external devices, providing

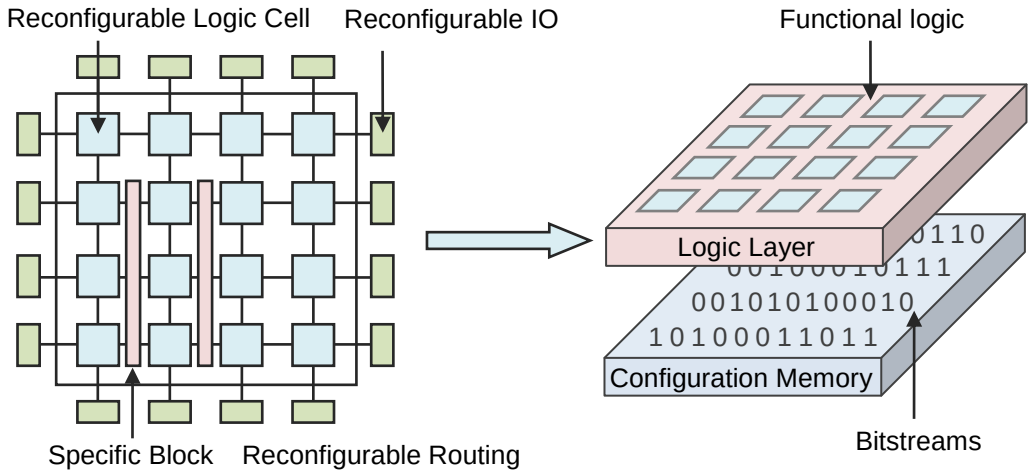


Figure 2.2 Generic FPGA Architecture

driving capabilities as well as physical protection from external hazards, such as Electrostatic Discharge (ESD). To further improve system performance and reduce chip size, specific blocks are integrated, such as BRAM blocks, Digital Signal Processing (DSP) blocks, and embedded processors. In a generic FPGA, most of the resources are reconfigurable since their functions are dictated by the content in the configuration memory, which is most commonly based on SRAM. Therefore, by writing data to the configuration memory, the chip functionality in the logic layer can be changed accordingly. The data in the configuration memory is usually called the bitstream, which records all of the static information of chip logic, such as static routings and combinatorial logic gates, and also the transient status, such as latched values in flip-flops and memory data. Bitstreams can be accessed externally through external configuration ports such as the Joint Test Action Group (JTAG) and boundary scan. For example, an FPGA can be configured by connecting the JTAG port with a PC, or a non-volatile memory on the same Printed Circuit Board (PCB). The bitstream is then be downloaded from either the PC or the non-volatile memory by the boot loader every time the FPGA is powered on. The above configuration is called static reconfiguration, where the whole chip has to be reset and reinitialized every time the configuration is performed. In effect, current DPR technology allows bitstreams to be loaded using internal chip logic at runtime and the chip can be partially reconfigured, leaving un-configured hardware uninterrupted.

DPR technology was firstly supported in the Xilinx XC6200 family FPGAs in early 1995. XC6200 FPGAs have a configuration data bus (8, 16 or 32 bits) and an address bus (16 bits), which allow the entire configuration memory to be programmed within 100 μs. In addition, a mask register is used to mask out certain bits in one configuration word, allowing parts of the configuration memory to be reprogrammed whereas the masked subsets remain unchanged, and therefore the device can be partially reconfigured dynamically [Churcher1995, Xilinx1997].

Xilinx Inc. subsequently further improved the DPR features in its following families, such as the VirtexII/ 4/ 5/ 6/ 7. These families support not only JTAG configuration, but also reconfigurability through two other ports: SelectMap and ICAP. The newly

supported configuration ports have significantly improved configurability in terms of both throughput and convenience [Xilinx2007, Xilinx2009, and Xilinx2010b]. In particular, ICAP allows the configuration memory to be accessed from its internal logic, which means that FPGA hardware can be internally self-reconfigured. Benefiting from the above, the configuration overhead is significantly reduced [Blodget2003, Xilinx2007, Xilinx2009]. For instance, the ICAP is located at the centre of the chip to give shorter and uniform propagation delay to all the registers. Since the internal port does not require any IO external to the chip, the ICAP bandwidth is enlarged to 32-bit to achieve higher configuration throughput. Moreover, the internal logic requires less voltage than external configuration ports, and hence power consumption is reduced [Xilinx2007, Xilinx2010b, and Xilinx2012a].

While Xilinx Corp. have developed a series of devices supporting dynamic reconfiguration, other FPGA vendors have also harnessed their configuration mechanism. For example, Altera Corp. enhanced reconfigurability in their products, such as the Stratix-V family FPGAs [Altera2010]. To date, the device support from FPGA vendors has substantially promoted DPR-associated techniques to reach a more realistic era for reconfigurable computing designs.

2.1.2 Design Flow Support

With the growth of DPR devices, Xilinx Corp. classified partial reconfiguration into two types, naming them difference-based and module-based DPR. Difference-based DPR is used when small changes are needed in hardware logic, such as changes of local parameters or logical equations. Difference-based DPR allows a design to be changed either from a user's original file (front-end), such as HDL files, or from the bitstream already generated (back-end). In module-based DPR, the whole chip area is divided into two partitions: a static region and partial reconfigurable (PR) regions (see Figure 2.3).

The static region is allocated to the system static logic, whose configuration does not change at run-time. PR regions are accommodated with PR modules, which are to be

reconfigured with different versions of bitstreams while the system is running. Module-based DPR reconfigures tasks as a whole instead of creating differences for particular functions, which gives better flexibility but requires more complicated

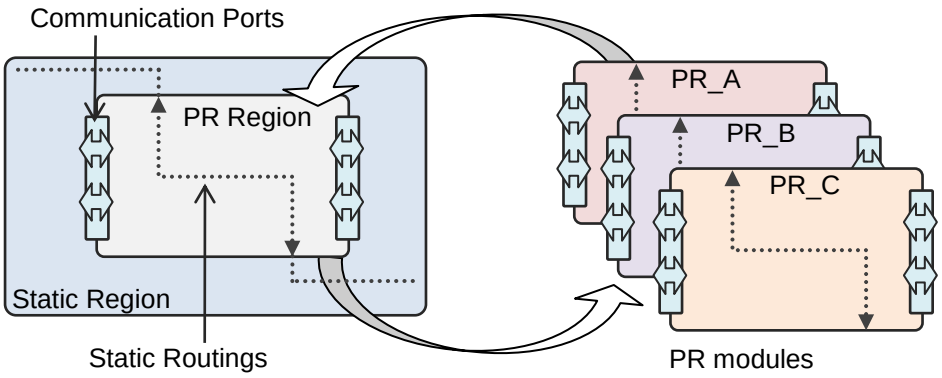


Figure 2.3 Modular-based Partial Reconfiguration

procedures. To enable regional reconfiguration, hardware modules are area-constrained within a dedicated square area, with communication ports accommodated at their boundaries. Note that routings in the static region could go across PR regions, and hence all of the PR modules have to be pre-synthesised with static routing (see Figure 2.3).

Based on the above theoretical concepts, FPGA vendors have developed their own design flows and solutions, together with integrated tools, software drivers and application demonstrations [Dye2010, Xilinx2010a, and Xilinx2010b]. For example, Xilinx Corp. released a partial reconfiguration user guide, in which the modular-based DPR design flow is demonstrated [Xilinx2010a]. Figure 2.4 illustrates the basic steps used to generate a full bitstream for three versions of partial reconfigurable modules (PR_A, PR_B, and PR_C).

First of all, five source HDL files have to be created by the users, including two static modules (top and static module) and three PR modules (PR_A, PR_B and PR_C). All the files are then synthesised into netlist format, such as an Electronic Data Interchange Format (EDIF) file or a Native Generic Database (NGD) file. After that, three netlist files (for the top module, static module and PR_A) are integrated together to implement a post place-and-route design, and to generate a full bitstream

(Full.bit) and a partial bitstream (A.bit). Afterwards, the implemented static module is copied and merged with two other PR modules to generate the other two partial bitstreams (B.bit and C.bit). To start the system, the full bitstream is used for first-

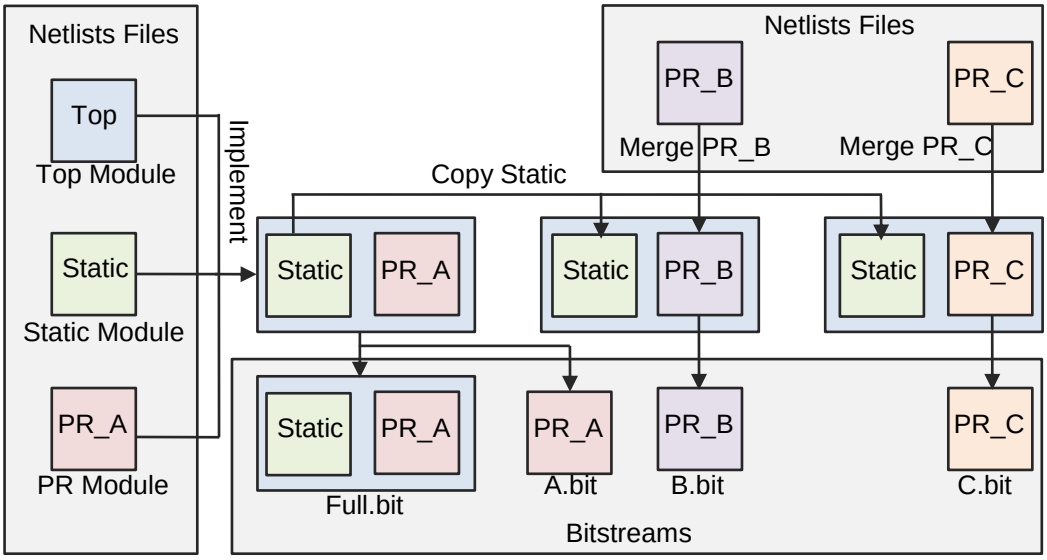


Figure 2.4 Overall PR Design Flow by Xilinx Corp.

time configuration, and then the other PR modules can be swapped within the PR region on demand at run-time.

To further advertise and commercialise this technique, Xilinx Corp. also integrated DPR features to its standard design suites, such as Xilinx PlanAhead and the Embedded Development Kit (EDK). In PlanAhead 12.x and later versions, the DPR design flow is supported, which provides the facilities to give area constraints for PR modules, as well as PR bitstream generation [Xilinx2012b]. In the EDK design suite, the ICAP can be instantiated, with the support from its hardware and software drivers [Xilinx2010b]. The current version of ICAP provides not only an easy-to-use API, but also higher performance with lower overheads. For example, in the Xilinx Virtex4 family of FPGAs, the ICAP supports 32-bit width with up to 100MHz operating frequency, which is capable of configuring a task in about hundreds of microseconds. In addition, Xilinx Corp. have proposed several applications based on DPR, such as a networked multiport interface, a dynamically reconfigurable packet processor and asymmetric key encryption, which apply DPR to enhance their performance [Xilinx2010a]. However, all of these approaches are only theoretically

proposed concepts, and are not sufficiently practical to be commercialised and industrialised. Although efforts have been made by industrial companies to support DPR applications, they do not provide a standard generic solution for applications, and hence the major development of DPR-based systems and applications still takes place in academic researches.

2.2 Hardware Management

With industrial hardware support, academic researcher aim to develop customer hardware tools with better flexibility and functionality. These hardware tools and methods can cope with the details of hardware and facilitate the development of innovative ROSs, where hardware and software can coexist in a seamless manner [Brebner1996, Wigley2001, Mignolet2003, and So2007]. To achieve this, various DPR hardware mechanisms have been proposed and developed using existing physical primitives, such as ICAP, and synthesis tools, like Xilinx Synthesis Technology (XST), to offer an easy-to-use Hardware Programing Interface (HPI) for later ROS development.

In order to improve programming flexibility, PR regions have to be partitioned with higher freedom. The existing commercialized PR flow shown in Figure 2.4 is based on a predefined PR region, where all the PR regions are floor planned in the design phase with fixed communication ports and static routings. This is also referred to as slot-based DPR [Carver2008, Koch2010, Al Farisi2011, and Korf2011]. In slot-based DPR, hardware resources are not highly efficiently used since the PR regions are predefined to cater for the biggest tasks, and therefore resources are wasted when a PR region is allocated to smaller tasks. In contrast, slot-less DPR uses an entire large PR region with no predefined boundary, and thus tasks can be placed anywhere, and resources are more efficiently used by geometrically managing the chip area at runtime [Koester2009, Becker2010]. Although slot-less DPR achieves better flexibility, it is more complicated and difficult to handle, since the static communication ports cannot be fixed to tasks. In regard to this, various researchers have proposed ideas and methodologies to cope with the complexities of slot-less

DPR, such as hardware heterogeneity [Becker2010], static routing [Bellato2004], and inter-task communication [Shayani2008].

2.2.1 Management of Heterogeneous FPGA Resources

With the increase in hardware integration, the management of hardware resources is becoming more complicated. In a generic FPGA, the hardware resources are not as homogeneous as they are in software memory space; on the contrary, hardware resources are of various resource types, such as configurable cells, memory blocks, DSPs, IO blocks and embedded processors, which are integrated into a single chip for better performance. Therefore, the same module used for a particular region cannot be arbitrarily mapped to other positions, and designers have to find a position where its resources exactly match the module. Some approaches have been proposed to quickly find an optimal place while maintaining better chip compactness and reducing chip fragmentation [Koester2009, Becker2010]. One such method is to keep shifting target tasks along with the chip resources until one suitable position is found [Becker2007]. Figure 2.5 gives an example of task shifting on a Xilinx FPGA.

The task is composed of two types of resource, Block RAM (BRAM) and Configurable Logic Block (CLB), and the FPGA has three types of resources, namely BRAM, CLB and Input / Output Block (IOB). To find a feasible position to allocate the target task, it is compared with the chip resources by shifting from left to right until a possible position is found. In general, localisability depends on chip regularity, which means that more homogeneously and regularly aligned resources can achieve better placement flexibility.

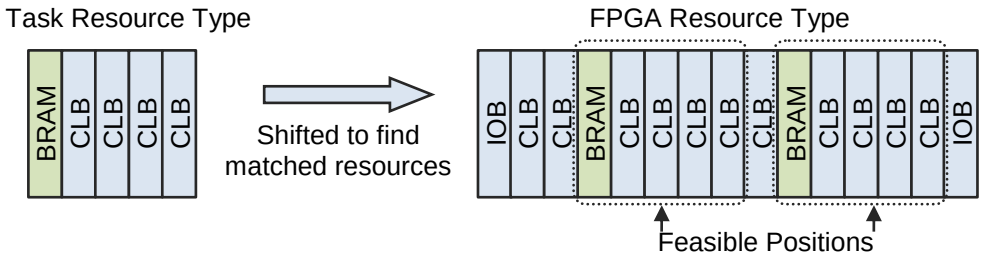


Figure 2.5 Task shifting to find matched positions

2.2.2 Preserving Static Routings

PR modules have to be aware of static routings. When implementing the full design and integrating the PR modules with the static module (see Figure 2.4), although PR modules are area-constrained within a dedicated region, this does not exclude routings being used by the static region. Therefore every PR module has to keep a copy of the static routing within itself. This causes conflict when modules are arbitrarily moved around non-predefined regions. Since the current Xilinx PR design tools do not support the PR module being implemented exclusively, researchers have developed their own tools to preserve static routings. One idea is to extract the static routing information by dynamically reading back PR modules at run-time and then XOR them with other PR modules before they are written back to the configuration memory [Sedcole2006]. However, although the primitive resources can be exclusively used, such as CLBs, BRAMs, DSPs, IOBs, there is no guarantee of the switch box routings [Bellato2004]. For instance, the switch box in Xilinx FPGAs uses Programmable Interconnection Points (PIPs) to arbitrarily link two or more connections. As long as two PIPs are used, the routing between them is automatically activated. Thus even if two modules are not using the same PIP, on undesired link could be activated when merging them together, resulting in a short circuit between the two signals (see Figure 2.6).

To avoid this conflict, a post-placed and routed module can be integrated, and thus the access to the particular switch box can be protected [Sedcole2006]. An alternative method was reported by Koch [Koch2009], who designed a blocker to prevent the use of certain routing resources by applying a “don’t touch” constraint. Similarly, certain areas can be preserved by placing an “anti-core” [Sohanghpurwala2011]. These methods usually rely on existing tools, such as XST,

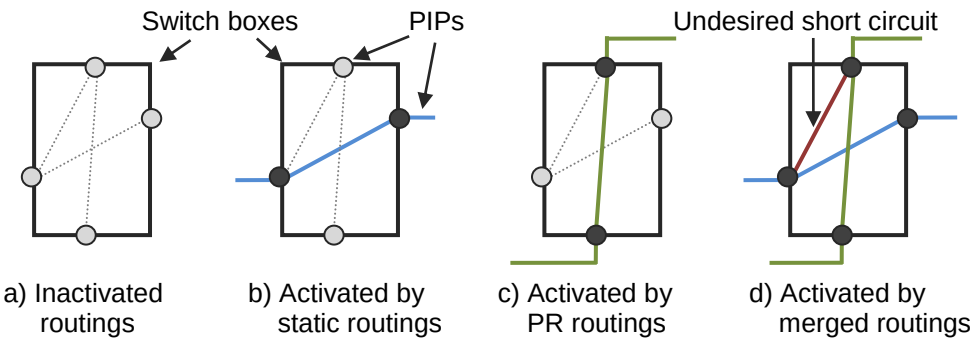


Figure 2.6 Routing conflicts caused by module overlapping

and solve their routing conflicts by using the "side-effect" of certain constraints, which could not be used as a generic solution applicable in other circumstances. Another disadvantage of such approaches is that they usually result in a less optimised design. For example, the block macros always take extra resources, and the preserved resources cannot be optimised by the synthesiser, leading to an inefficient overall design.

2.2.3 Inter-Task Communication

Although slot-less DPR improves the flexibility in task placement, it also causes difficulties in communication between PR modules. Since the PR modules have to be placed at arbitrary positions, the communication ports, as well as inter-task routings have to correspondingly be updated with changes in the positions. The updating of the inter-task routings or the communication network is achieved by reconfiguring the FPGA routing resources. Depending on the reconfiguration effort required, the reconfigurable inter-task communication mechanism generally falls into two categories: 1) on-line routing generation, which generates routings or communication modules to build links between tasks; and 2) a Reconfigurable Network on Chip (RNoC), which relies on a relatively fixed communication infrastructure with reconfigurable routing tables.

1) On-line routing generation can connect two or more tasks for on-demand communications on-the-fly by using existing FPGA resources, such as switch boxes or Look-Up-Tables (LUTs) (see Figure 2.7). In such approaches, the tasks are

integrated with a fixed communication port, which is used to connect adjacent resources for communication. The routing resource can be implemented using the PIPs in switch boxes to directly link from source to destination without internal registering, which gives zero latency but longer propagation delay [Suris2008, Couch2011]. Alternatively, CLBs can be used to build Routing Primitives (RPs) for both horizontal and vertical connections, where signals can not only be registered by flip-flops, but also multiplexed by the truth table in the LUTs [Hubner2006, Upegui2006, and Shayani2008]. The latter CLB-based process provides better reconfigurable flexibility and controllability for the routings, at the expense of larger resource consumption. On the other hand, the routing topologies can be calculated through predefined routing models, or simplified routing algorithms. The latter is based on real-time computation to calculate an optimised routing, which can obtain better efficiency using less memory space. To better support on-line routing topologies, the analysis of timing constraints, such as maximum clock rate for particular routings can also be dynamically calculated through simplified algorithms, giving a more reliable design [Koch2010a, Koch2010b]. Based on these concepts, some practical applications have been implemented to demonstrate on-line routing, such as Software Defined Radio (SDR) [Suris2008]. However, on-line routing generation also has disadvantages. For example, the long path of the routing increases either the propagation delays or the transmission latency, resulting in worse performance. The chip area is also more fragmented due to long path routing thus reducing the available compact free areas for allocating upcoming tasks. Moreover, both predefined routings and real-time algorithms consume large amounts of resources, as the predefined routings require large memory space to store the predefined routings bitstream, and on-line algorithms always require an embedded processor for heavy computation [Jara-Berrocal2010, Silva2010].

2) The reconfigurable network on-chip uses a relatively fixed static architecture where its routing tables can be reconfigured, instead of arbitrarily placing routing-modules. For example, a Reconfigurable Multiple Bus (RMB) has been designed to build an inter-task communication network [Ahmadinia2005] (see Figure 2.8). In

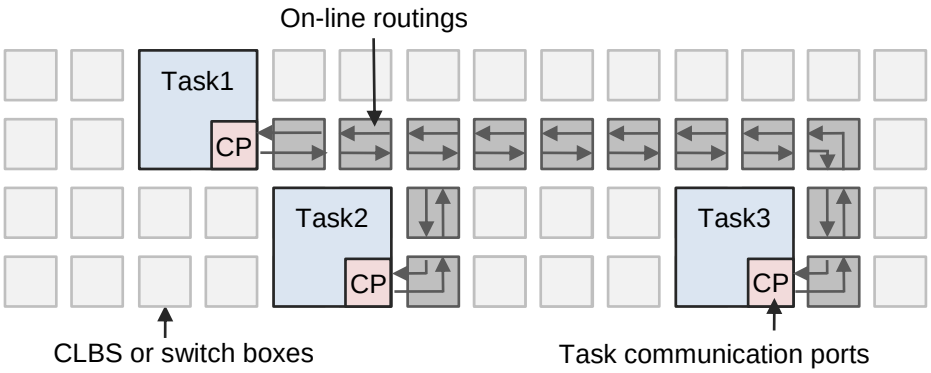


Figure 2.7 On-line routings generation

it, the entire chip resource is partitioned into separate slots used for accommodating upcoming tasks. Each slot is connected to a cross-point router, which controls data coming from the connected slots and routes it to the destination slot address. In such an approach, not only can the PR modules be reconfigured in the slots, but also the RMB can be reconfigured to adopt different routes. This approach reduces the conventional routing overhead and improves resource efficiency. In a similar approach, the system consists of a fixed network infrastructure [Palesi2007], with extra reconfigurability used to adopt different routes according to current running modules. Alternatively, a packet switching mechanism has been proposed [Sedcole2007], in which the data packet is passed through a predetermined fixed bus system. A combination of both packet switching and circuit switching (communication modules) was introduced [Stensgaard2008], where more physical connections can be created for communication-intensive tasks. Note that the above reconfigurable networks on-chip still rely on a number of static routings, along with slot-based module reconfiguration; thus they considerably limits the reconfigurable flexibility of the FPGA. To reduce the area overhead associated with the communication network in the forms of minimising routing resources, a novel solution has been proposed [Shelburne2008, Sander2008], which harnesses the configuration port and uses it for route-less communication. For example, data can be read-back from the sender module through the configuration ports, and then written back to the receiver module. Although such approaches significantly reduce

communication overheads, they do not support concurrent communication for multiple tasks at the same time.

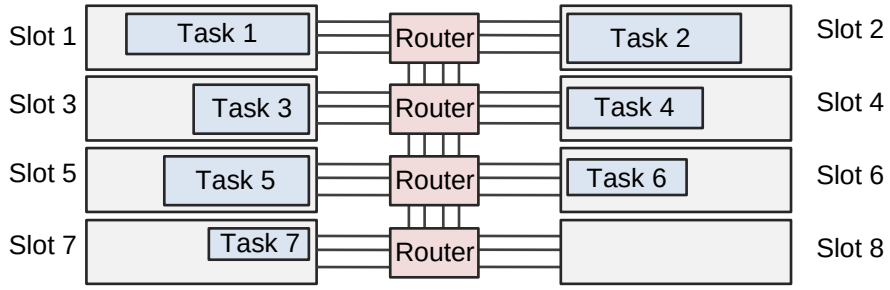


Figure 2.8 Reconfigurable network on chip

2.2.4 DPR Tool Integration

Although various solutions have been proposed to cope with particular hardware features, they do not provide an integrated development environment to automatize DPR design. To further facilitate reconfigurable hardware resources, an integrated developing platform needs to be provided to support the whole design flow, from single module RTL synthesis, via whole system integration and simulation, to a front-end Graphic User Interface (GUI). To date, only a relatively small number of approaches have managed to complete an integrated platform supporting the whole design flow. The PlanAhead PR-flow is one such integrated development tool chain commercialised by Xilinx Corp., which has been integrated in the Xilinx design suite [Xilinx2010a, Xilinx2012b]. However, it only supports slot-based DPR, in which most of the reconfiguration relies on a fixed infrastructure with limited reconfigurable flexibility. To support slot-less reconfiguration with more flexibility, an integrated platform called GoAhead has been developed to provide an easy-to-use interface while maximally exploring reconfigurability [Koch2008]. This is based on the ReCoBus, which is a communication infrastructure previously developed by the same author [Koch2008]. The ReCoBus leverages Xilinx synthesis and implementation tools to generate PR modules exclusively. Instead of constraining a module to a particular region, it uses a blocker macro to prohibit using logic resources outside of the task region. By preventing the use of external logic, the task

can be constrained within its own area without spreading its signals elsewhere. Moreover, the GoAhead platform supports a GUI, which gives designers a more friendly interface and better productivity. Nevertheless, GoAhead does not fully support a high-level simulation tool for whole system behaviour. In effect, most of the approaches so far rely mainly on separated RTL simulations for each module individually, and the overall system simulation is usually achieved by statically replacing different PR modules in the whole design [Raabe2008]. To give run-time simulation support and overall system evaluation, tools have to be better integrated with the front-end interface, such as an OS-like API, to ease real-time debugging and simulation.

2.3 Reconfigurable Operating Systems

An operating system is a collection of software built upon processors and other hardware to provide a standard programming interface for applications. It provides basic services to facilitate user application development, such as memory management, device driver management, basic I/O, interrupt, file systems and networks etc. The concept of the Reconfigurable Operating Systems (ROS) was first proposed by Brebner [Brebner1996], who suggested that the OS can be built on reconfigurable hardware, such as FPGAs, and thus virtual hardware resources can be used by sharing the same hardware among different tasks in a time-multiplexed fashion, which is similar to the operation of virtual memory in software programming. To achieve this, the ROS has to provide not only the capability of memory management, but also the control of reconfigurable hardware resources [Wigley2001]. For example, in the software based OS, a software task is created and allocated to a free memory space and scheduled among other tasks. If there is no available memory space, heterogeneous data storage such as disk storage will be used. From a hardware perspective, if a hardware component such as a multiplier is required but out of order, the system will free (or preempt) parts of its other hardware resources which are not currently occupied, and use them to reconfigure the required component; thereby, more virtual hardware resources will be available for use.

Figure 2.9 gives a comparison between virtual memory and virtual hardware resources. As with software memory space which can be mapped to heterogeneous

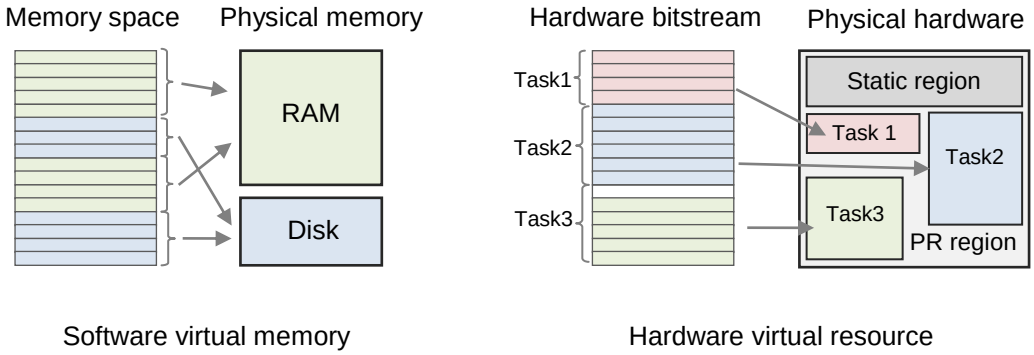


Figure 2.9 Virtual memory and virtual hardware

physical resources, the hardware components stored in the bitstream library can be mapped to FPGA resources in the PR region. Since hardware tasks can be preempted and freed from the resources, more virtual components will become available by sharing hardware resources in a time-multiplexed fashion.

Compared with a conventional OS, the most significant feature of the ROS is that it supports extra services for hardware resource management, which essentially requires dynamic hardware scheduling and allocation on the run [Wigley2002]. In order to meet system timing constraints meanwhile using resource more efficiently, tasks have to be scheduled in real-time and allocated to optimised positions, whereby task deadlines can be met and chip fragmentation can be reduced respectively. The following sections summarise some previous approaches used in hardware task scheduling and allocation, and then briefly describe some representative ROSs.

2.3.1 Hardware Task Scheduling

In a real-time environment, such as an event-triggered system or a time-triggered system, tasks have to meet different levels of real-time constrains in order to provide a good Quality of Service (QoS) [Buttazzo2004]. Generally, the real-time feature of a task can be defined using the following variables: task releasing time r_i, computation time $T_{C,i}$, and relative deadline D_i (see Figure 2.10). The task releasing

time is the absolute time when the task is triggered or required and the computation time is the relative time period from task starting to task finishing. If there is no available resource after a task is released, the task can be put on hold to wait for other resources to be freed. However, the task cannot wait for more time than its relative deadline D_i, otherwise it becomes obsolete. In the proposed hardware environment, the task computation time consists of the task allocation time and task execution time. The task allocation time refers to the time spent on reconfiguring a task to the chip, and it is proportional to the number of logic resources it uses. Likewise, the task execution time is the product obtained by multiplying the task clock cycle with the system clock period. In most software OSs, tasks are assigned with a priority value which is used to decide task running order and task preempting. In a real-time OS, the priority is usually decided according to task deadlines. For example, in the first proposed Deadline Monotonic (DM) system [Leung1982], a task with the shortest relative deadline is always assigned with highest priority. Alternatively, the more commonly used Earliest Deadline First (EDF) algorithm gives the highest priority to tasks which have the earliest absolute deadline [Liu1973]. The EDF gives dynamic scheduling and hence achieves better adaptability to the environment, whereas DM is statically determined off-line and therefore results in less scheduling time overhead.

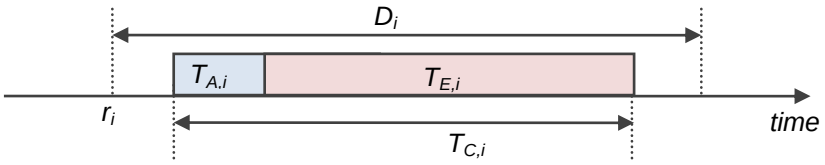


Figure 2.10 Hardware task time model

On the other hand, scheduling algorithms can also be categorised into preemptive and non-preemptive scheduling. In a non-preemptive scheduling system, hardware tasks cannot be interrupted before they finish their computations, which gives more predictable performance and less scheduling overhead [Steiger2003, Cui2007]. In a preemptive system, tasks can be halted and preempted when their resources are

required by tasks with higher priority [Danne2005]. The preemptive system provides better adaptability and real-time performance for critical tasks, at the expense of management simplicity and scheduling speed. For example, the context switching of hardware requires a number of steps, such as flip-flop status capturing, bitstream read-back, task de-allocation, and task reconfiguration. Hence it is much more complicated than software context switching, where only registers need to be stacked [Kalte2005, Jozwik2010]. Although hardware context switching has been considered [Simmler2000, Jovanovic2007, Ahmadinia2004a], it is still not practical since the limited throughput of configuration ports results in large time overheads.

Alternatively, scheduling efficiency can be improved by taking into consideration other task effects, such as task area consumption, task dependencies and inter-task communication. Task area consumption refers the task size and its position, which can be analysed together in combination with the time constraint to give three-dimensional real-time scheduling [Lu2009, Marconi2010]. Moreover, by properly aligning tasks according to their dependencies and communications, more efficient scheduling and a smaller communication footprint can be achieved [Redaelli2009, Lu2010, and Gohringer2011].

2.3.2 Hardware Task Allocating

In a software OS, an efficacious task allocator can choose the best memory segment to allocate the target software task to, whereby the physical memory space is less fragmented and thus more efficiently used. Likewise, in ROSs, hardware tasks can be allocated to optimized positions so as to reduce chip fragmentation, leaving more compact hardware resources available for later tasks. However, since the hardware resources are two-dimensional, hardware allocation is more complicated than software allocation in one-dimensional memory space. Therefore, the two-dimensional chip area is usually analysed geometrically using topological methodologies, and the 2D-packing algorithms can be applied for area management. An efficacious hardware allocation algorithm can reduce chip fragmentation and retain more compact areas, meanwhile requiring less computation time. Towards this

goal, a number of algorithms have been developed for placement optimization, which can be differentiated according to the way they track unoccupied chip resources.

Various algorithms have been considered to describe and analyse the status of a chip's free area. In the pioneering work proposed by Bazargan [Bazargan2000], the compact area is indicated by Maximum Empty Rectangles (MER). In this context, the whole chip is modelled as a rectilinear grid formed by cell arrays to facilitate matrix computation. After placing upcoming tasks, the whole chip area is fragmented into several squares. These squares are combined together to form MERs, and all MERs can be used to allocate later tasks. For example, if a task is to be allocated, the task size will be compared with each MER and then the allocator will choose to use either the first feasible MER found to place the task, or the smallest MER that is possible for the task. The former is called the First-Fit (FF) decision, which gives faster computing speed; whereas the latter is named the Best-Fit (BF) decision, which gives better area optimization but needs longer computing time. One disadvantage of MER-based algorithms is that they require heavy computation and large memory space, since the number of MERs increases dramatically with the insertion of new tasks. Therefore, methods have been proposed to reduce the memory overhead used for tracking free areas. For example, the partitioned MERs can be stored and managed by a binary tree-like structure for faster searching and manipulation [Morandi2008]. Another author's idea [Ahmadinia2004b] is to track the occupied area instead of free areas, which reduces computation time and saves memory space. Alternatively to MER, chip status can be analysed by a set of vertex lists which describe the contours of area fragments [Tabero2004, Ahmadinia2007]. Compared with rectangle-based algorithms, vertex lists require less memory to store area information, but more time is consumed to find possible positions. In addition, in some other approaches, the chip is modelled as geographical blocks, such as staircases or line arrays, which eases area analysis at the cost of placement granularity and flexibility [Handa2004, Cui2007].

On the other hand, the behaviour of a real task is considered to improve placement efficiency. For instance, highly coupled tasks are prioritized and placed at adjacent

positions to reduce the communication footprint [Ahmadinia2004b, Morandi2008], whereas tasks with less interaction can be placed with lower priorities. Besides this, the task execution time is equally important in dynamic placement. For example, more expensive resources can be preserved for tasks with shorter execution time, and therefore they can be shared among more tasks. By considering time effects, the hardware tasks can be modelled as three-dimensional cubes to be packed into the entire chip volume. Combined with efficient scheduling algorithms, the 3D packing heuristic gives considerably higher efficiency although more memory and time are consumed [Tabero2006, Marconi2010].

In addition, the fragmented hardware resources can be defragged dynamically by reallocating already placed tasks, which is an emulation of the software defragmentation tool that rearranges file blocks on disks to make them contiguous [Ejnioui2005, van der Veen2005]. However, to do this, multiple tasks have to be reallocated concurrently, which requires bitstream reading back, algorithm computation, bitstream changing and task writing back. These procedures introduce a considerable number of reconfiguration overheads; hence, most of them are not practical to be used in real-time [Li2002, Kalte2005, and El Farag2007].

2.3.3 Completed Reconfigurable Operating Systems

A completed reconfigurable operating system should at least be capable of hardware resource management, hardware task scheduling and allocation, and providing an OS-like API. To date, most such approaches are built on Unix-like OSs (such as Linux) and some of them support multi-core programming interfaces, such as Message Passing Interface (MPI). This significantly increases programming productivity and design portability. In these systems, although tasks are implemented using different mechanisms such as a hardware state machine, host processor, or reconfigurable coprocessors, they usually have the same look and feel at the user programming interface.

In early 2003, the idea of multiple task implementation was proposed, indicating that tasks can be implemented in both hardware and software, or a mix of both. Inspired

by this idea, a task can be executed in either software or hardware and switched from one to the other depending on environment requirements [Mignolet2003]. This concept has since been extended in later years, and a system has been proposed [Zhou2005] allowing hardware and software tasks to be instantiated from a generic API using an identical function call [Zhou2005].

In 2005, a slot-based ROS was developed on a Xilinx Virtex-II FPGA, in which hardware tasks could be reconfigured within resource slots [Walder2005]. The communication between slots is implemented on FIFO-based control logic fixed at the top of each slot. In this system, tasks can be scheduled and allocated at any of the slots. In addition, to allocate tasks with sizes larger than the slot, multiple slots can be combined together to make a larger area possible to accommodate the task. Also in the same year, HybridThreads was designed to run multiple threads on both software and hardware simultaneously on a hybrid system consisting of both processors and FPGAs [Andrews2005]. However this approach does not support task reallocation and therefore resources are less efficiently used since they are fixed to particular tasks.

In 2007, an ROS composed of multi-FPGAs was developed in the University of California, Berkeley, called BORPH (an operating system for FPGA-based reconfigurable computers) [So2007]. It consists of five Virtex-II Pro FPGAs connected by a point-to-point, bi-directional, 8-bit bus running at 50MHz. The same bus is used for both configuration and inter-FPGA communication. One of the five FPGAs is called the control-FPGA or host, which controls all peripheries implemented on the other four FPGAs (the user-FPGAs). The host is developed on a Linux based software kernel implemented on a PowerPC 405 microprocessor, offering a UNIX-like API [So2008]. All the user-FPGAs are reconfigurable externally through a SelectMAP port, which is connected with to bus and mastered by the control-FPGA. However, the high heterogeneity of hardware resources makes it difficult to apply efficient scheduling and allocating algorithms and thus they are not supported in BORPH.

The following year, a real-time ROS was developed on a single FPGA chip (Virtex-II or Virtex4), called ReconOS [Lubbers2010]. In ReconOS, tasks can be reconfigured within predefined slots, and a logic controller is used for synchronisation and data-exchange between different slots. In particular, the system allows tasks to be reconfigured using the internal configuration port, which significantly improves the configuration speed. In the same year, a ROS called CAP-OS was designed on a Virtex4 FPGA with the support of the Message Passing Interface (MPI) [Gohringer2010a]. However, despite the fact that the CAP-OS takes into consideration task dependencies to improve scheduling efficiency, the current prototype only supports software execution [Gohringer2010b].

In 2011, a shared memory based ROS was developed on Virtex5 FPGA, called FUSE (Front-end user framework for O/S abstraction of hardware accelerators) [Ismail2011]. This is a slot-based ROS with Unix-like API programmed on a Xilinx MicroBlaze soft-core microprocessor. In FUSE, hardware tasks can easily be accessed from software by using shared memory and global mapped memory space. Benefiting from this, not only is the communication overhead reduced, but also programming productivity is improved.

The R3TOS project (the present system of this thesis) was initialized in 2008, carried out within the System Level Integration Group (SLIG) at the University of Edinburgh (UK), in collaboration with IKERLAN-IK4 Research Alliance (Basque Country, Spain). By the time the author joined R3TOS project (in 2010), R3TOS has been developed for over two years, during which the project prototype was initialized, such as the overall system architecture and a preliminary version of the scheduling and allocating algorithms. The real implementation and integration started from 2010, and the work contributed by the author during the three years are presented in this thesis, which will be detailed in the following chapters.

2.4 Conclusion

This chapter has reviewed previous representative works related to DPR techniques, which include industrial supports, tools and methods for hardware management, and the development of the ROS. The industrial companies played a leading role in this development by providing fundamental hardware supports such as ICAP and frame-based bitstream. Following that, various customer tools and methods have been proposed to cope with complicated low-level DPR hardware and to provide HPI for upper level design. With such support, ROSs have been developed, which give an OS-like API to facilitate user application development. It is noted that most of these approaches do not take full advantage of modern DPR, such as the high programming flexibility and fine-grained configuration offered by currently available FPGAs.

Chapter 3

R3TOS System Overview

The Reliable, Reconfigurable Real-time Operating System (R3TOS) is a slot-less reconfigurable system that exploits the reconfigurability of modern FPGAs in an innovative way, whereby both fault tolerance and real-time performance are improved compared with non-reconfigurable systems. It bridges the gap between the design of high-level reconfigurable applications and the management of the low-level hardware implementation, and gives application designers an OS-like development environment. First of all, unlike with slot-based reconfiguration, slot-less reconfigurability gives the R3TOS the flexibility to configure FPGAs in a fine-grained granularity, which effectively increases resource efficiency and reusability [Hong2011b]. Moreover, the R3TOS supports multi-user, multi-tasking applications, where multiple application tasks from different users can be customized at runtime and are executed in parallel [Hong2013b]. Furthermore, the R3TOS is capable of detecting, isolating, and circumventing any transient error or permanent damage by means of reconfigurable techniques [Iturbe2009]. Last, but not least, the R3TOS schedules both software and hardware tasks according to real-time criteria, providing a good quality of service [Iturbe2013b].

This chapter outlines the R3TOS's basic functioning model and main features, prior to the detailed designs and implementations presented in subsequent chapters. This

Related publications: [Hong2011b], [Hong2013b] and [Iturbe2013b].

chapter first outlines the system's basic operating model, in which a simplified system prototype is illustrated to give a straightforward overall understanding of the R3TOS. After that, the system operation mode is explained in three phases: 1) the task generation phase where tasks are designed, 2) the system computing phase where they are executed, and 3) a fault toleration phase where tasks are recovered. The work presented in this chapter was developed in cooperation with the author's colleague Xabier Iturbe, and some of the ideas discussed have been previously published [Hong2011b, Hong2013b, and Iturbe2013b].

3.1 Basic Operating Model

In the R3TOS, hardware tasks behave in a similar way to software tasks, which can be called from the user and scheduled, allocated, and mapped to the chip resources. During system operation, the hardware tasks can be swapped in/out of the chip flexibly without boundary concerns. Figure 3.1 gives a simplified overview of the R3TOS task operating model. Here, user-defined tasks are firstly placed in a task queue, ready to be scheduled and allocated by the task scheduler and task allocator respectively. The task scheduler schedules tasks in real time and passes the information about the prioritized task to the task allocator. Then the task allocator investigates the entire chip resources in order to search for all possible locations that are free and matched with the task resource requirements. Afterwards an optimum location is selected in which to place the task, whereby more compact free areas can be retained and chip fragmentations can be reduced. Tasks can start being executed as soon as they are configured to the chip through the ICAP port. Once a task is finished, it can either stay in the allocated stage to be reactivated again for another computation, or be de-allocated so as to free the hardware resource for upcoming tasks. In order to guarantee a reliable computing environment, the system uses a scrubbing technique to periodically read back the configuration memory in order to detect any occurrence of error or damage. By applying scrubbing, transient errors can be mitigated by reconfiguring the faulty region, and unfixable resources will be marked as permanent damage whose use should be avoided.

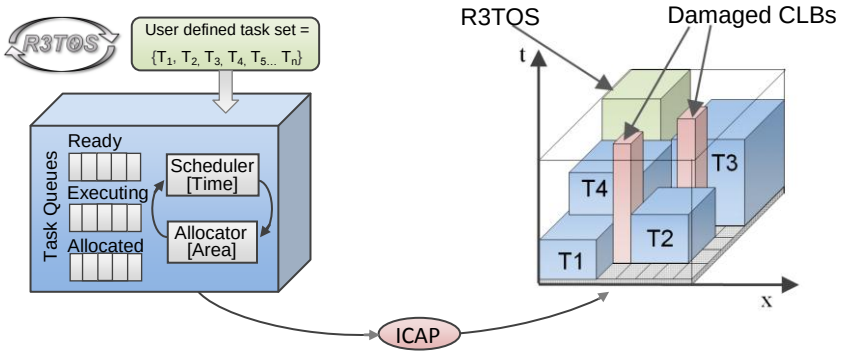


Figure 3.1 R3TOS task operating model

In the R3TOS, the FPGA chip is divided into two regions: a static region and a PR region. Unlike in slot-based systems, the PR region is not partitioned into predefined slots but rather acts as a whole PR region in which tasks can be configured anywhere. The kernel of the R3TOS system is located in the static region, and PR tasks can be dynamically allocated in the PR region according to user demands. A simplified system prototype is given in Figure 3.2, in which the basic components and functions are shown. The system consists of an R3TOS hardware microkernel (HWμK) and a main CPU for the application API. The HWμK is responsible for all hardware management, and the CPU is a generic processor providing a standard API for user applications. There are three main components central to the HWμK,: task scheduler, the task allocator, and ICAP manager. These are connected together using an internal system bus (see Figure 3.2).

In this prototype, all of the HWμK components, namely the scheduler, allocator and ICAP manager, are implemented individually on three PicoBlaze processors. In order to efficiently schedule tasks in real-time and to exploit hardware resources at runtime most efficaciously, novel scheduling algorithms and allocating algorithms have been developed, which are both coded in the assembler and implemented in the task scheduler and allocator respectively. The main CPU program is implemented on the Xilinx softcore microprocessor MicroBlaze [Xilinx2002] in the current version, and it can be ported into other FPGA based processors such as PowerPC or ARM Cortex-A9. In the HWμK, the scheduler has a task memory, which is implemented in a BRAM to store all the task information. Likewise, the task allocator uses a matrix

memory to

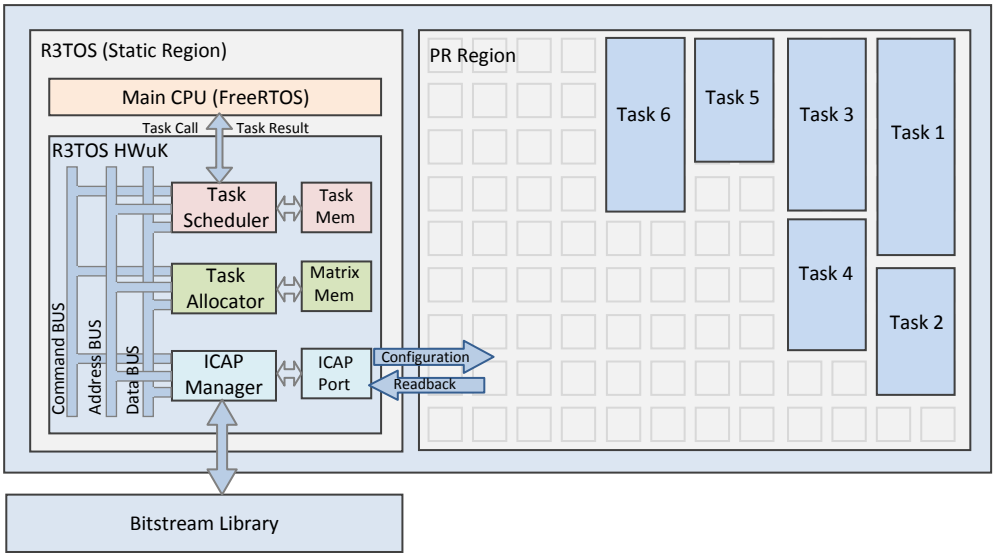


Figure 3.2 Simplified R3TOS architecture

store information about the chip area, which is modelled as a rectilinear grid matrix and stored in another BRAM. After it has been decided to place a task on the chip, the ICAP manager will receive the task ID and location from the allocator. It then uses the task ID to fetch the task's bitstream from the bitstream library. After updating the bitstream with the new task position, the bitstream is downloaded to the configuration memory through the ICAP. The ICAP manager is also responsible for inter-task communication, delivering the producer task's output to the consumer task's input.

There are some direct advantages which can be achieved by using this computing model, including reconfiguration flexibility, resource efficiency, fault tolerance, power efficiency, high performance and prolonged device lifetime. First of all, this architecture provides free chip space without being fragmented by static routings. Therefore tasks can be arbitrarily placed anywhere on the chip. As a result, logic resources can be used more efficiently by managing areas with a finer granularity [Hong2011b]. In addition, benefiting from high reconfigurable flexibility, fault tolerance can be achieved by reconfiguring task regions or reallocating tasks away from the damaged resources. For instance, a transient soft error can be mitigated by reconfiguring the task region with the original bitstream, and the damaged resources

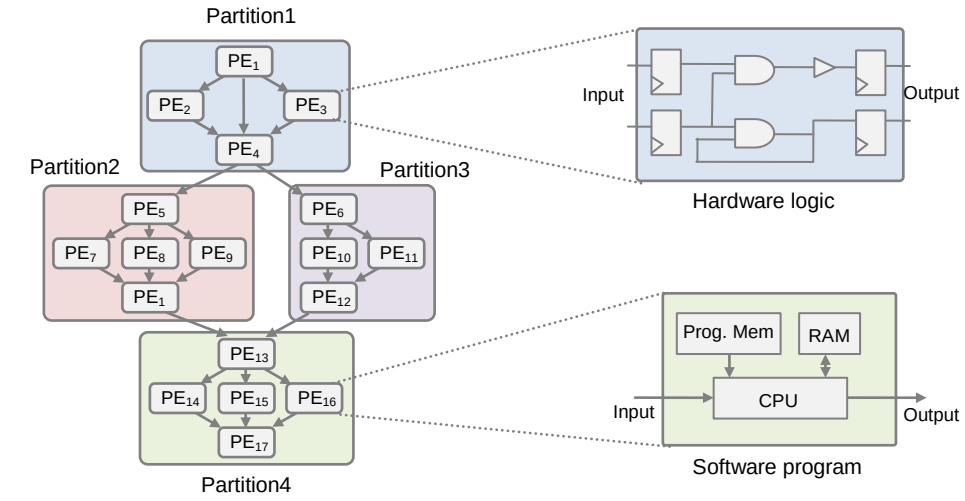
can be avoided by moving tasks to another position [Iturbe2009]. Moreover, power efficiency is improved since unused tasks can be removed from the chip [Zhang2006]. Furthermore, as tasks are area-constrained within their dedicated regions, the propagation delay is reduced and thus a higher clock rate can be applied and task performance can be improved [Iturbe2012]. Last, but not least, the hardware resources are not dedicated to any particular task, but shared among all tasks. As a consequence, device lifetime is prolonged since computing density is uniformly distributed around the whole chip [Srinivasan2006, Feng2010, and Angermeier2011].

The following sections further illustrate the task design flow and system operation in the three different phases; namely, the task design stage, the system operating stage, and the fault detection and recovery stage.

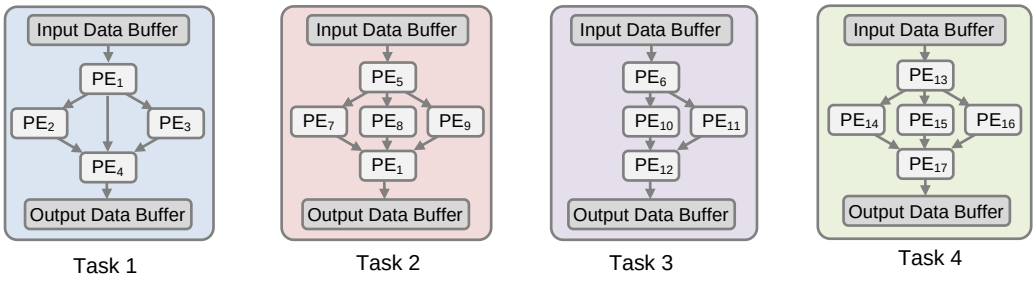
3.2 Application Task Design

In the R3TOS, an application program can be co-designed in both hardware and software. The software design is similar to the traditional ROS API programming, whereas the hardware needs to be firstly partitioned into less-coupled subsections according to its computing dependency, and then wrapped with Task Control Logic (TCL) to enable inter-task communication. Afterwards, wrapped hardware tasks are registered in the main CPU, where they can behave just like pure software tasks from the user's viewpoint.

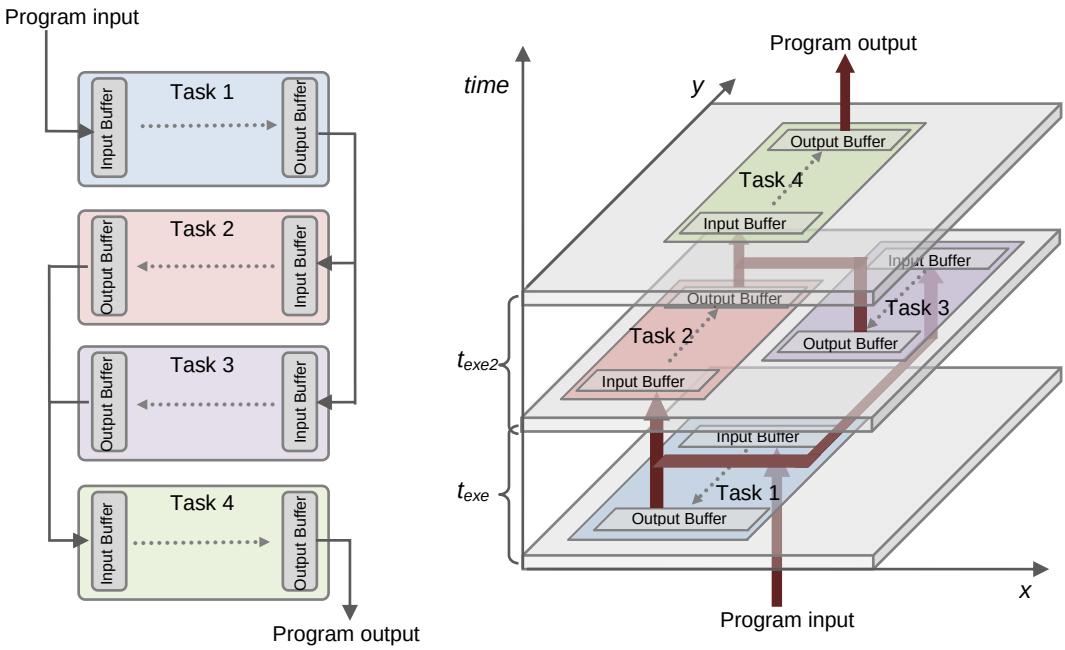
Figure 3.3 gives the general steps used to generate and execute an application task in the R3TOS. To implement the application program on hardware, the program is firstly partitioned into subsections according to its dependency graph (see Figure 3.3.a). A good quality of partitioning can reduce the longest datapath, whereby the maximum operating clock frequency can be increased and thus the performance is improved [Bertels2011]. In this example, the whole application program is composed of 17 processes which are mapped to a set of Processing Elements (PEs) from PE_1 to PE_{17}. Each PE can be implemented using different computing resources, such as pure hardware logic, software processors, or a mixture of both. Highly coupled processes



a) Program partitioning



b) Wrapping with task control logic



c) Task execution

Figure 3.3 Design and execution of application tasks

are grouped together to make one partition. Consequently, partitions are relatively independent and thus require less data exchange, whereby the overhead of inter-task communications can be reduced. After partitioning, each partition is wrapped with task control logic (TCL) to build a full task. The task control logic includes an Input Data Buffer (IDB) and an Output Data Buffer (ODB) for communications with other tasks (see Figure 3.3.b). In addition, some synchronization mechanisms, such as hardware semaphores, are also integrated within each task, to synchronize multi-task operations. Figure 3.3.c gives an example of task execution during system operation. According to its task dependency, *task1* is firstly configured to the chip and the program input is sent to its IDB. After *task1* finishes its execution, it is removed from the chip and *task2* and *task3* are subsequently placed. The result from *task1* is buffered in the FIFO as are the inputs of both *task2* and *task3*. Likewise, *task4* is allocated after *task2* and *task3*, to produce the final result of the program. Communication between tasks is achieved by using our novel inter-task communication mechanism, which is introduced in section 3.3.2.

The hardware management requires complicated scheduling and allocating routines. However, they are automatically handled by the R3TOS hardware kernel. Hence, from a user's viewpoint, all hardware tasks behave like software tasks, which can be called from the main CPU and return to the result of the function. In order to be compatible with the software environment, every hardware task has a software version image, which describes the software features of a hardware task such as function calling, arguments passing and result returning. Therefore, hardware tasks can be called from the main CPU in the same way as in traditional software programs, but they are indeed executed in real hardware logic. These provide genuine computing parallelism as well as the high performance of dedicated circuits.

Figure 3.4 gives the overall design and execution flow of tasks in the R3TOS. First of all, application designers can co-design their application in either hardware or software. The hardware can be programed using HDL language or the synthesised

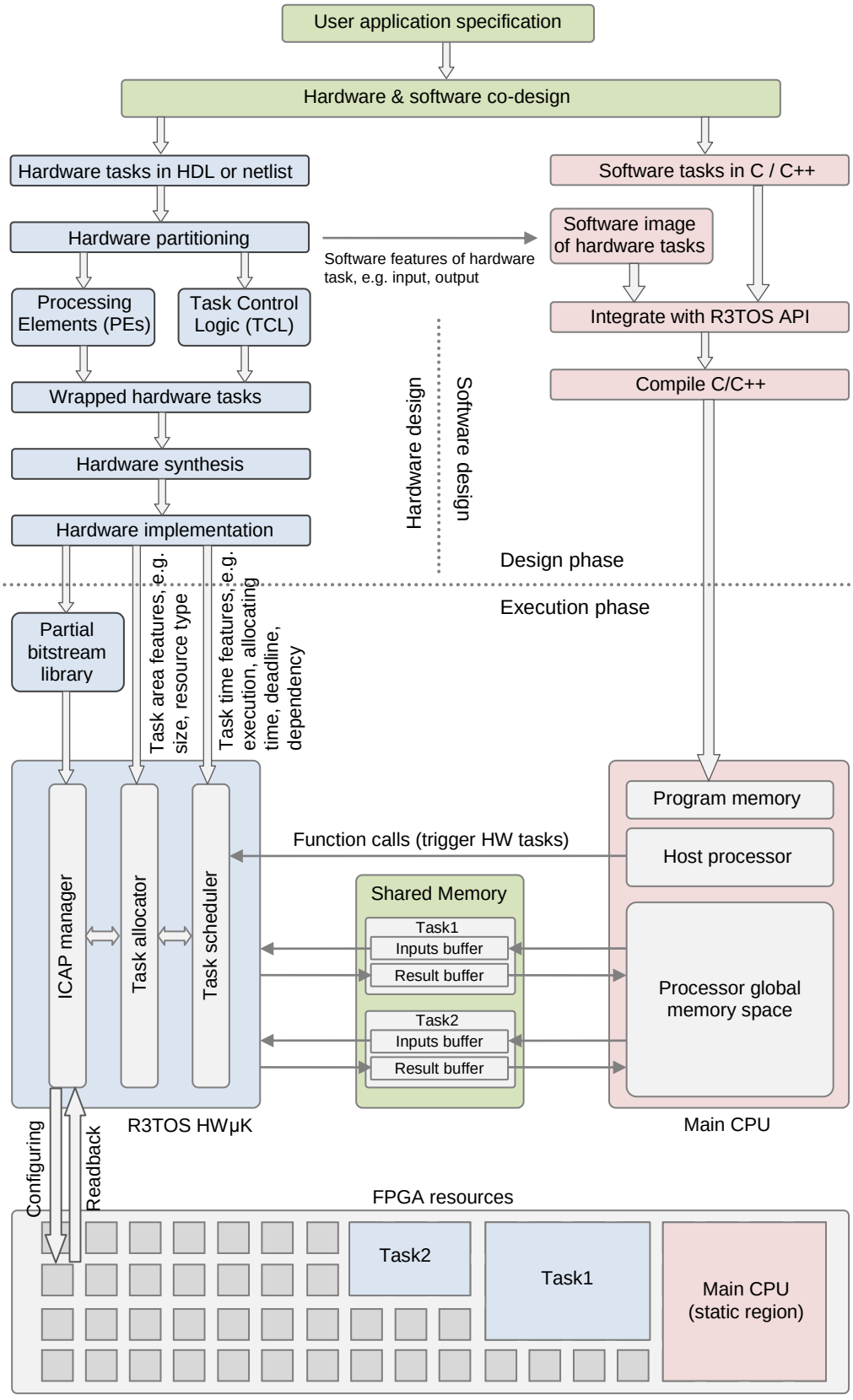


Figure 3.4 Hardware & software co-design of a hardware task

netlist files and the software can be coded in the standard C language (ANSI-C) or MicroBlaze assembler language. Afterwards, the subsequent hardware and software design flows are relatively distinct although some interaction is required.

The hardware program needs to be partitioned into separate tasks, which are loosely coupled with each other in order to reduce the inter-task communication. The software features of the partitioned tasks are extracted so as to be used in the main CPU, such as task inputs and outputs. Afterwards, the partitioned tasks are wrapped with task control logic to give a fully functional task capable of communicating with either the main CPU or other tasks. The wrapped tasks are then synthesised and implemented to generate partial bitstreams, which are stored in the bitstream library to be used for runtime reconfiguration. During synthesis, certain design constraints are used to generate tasks within a particular area, excluding the static routings, by using currently available synthesis tools such as Xilinx XST and PlanAhead. After synthesis and implementation, the task timing features and area features are obtained and saved in the task scheduler and task allocator respectively. The timing features are used for real-time scheduling, and include the task allocating time, task executing time, task deadlines, and task dependency. The task area features refer to task size and resource types, which are to be used to find a possible chip location at runtime.

The software design flow is similar to that in software application task programming in a traditional real-time OS API, in which the application tasks are programmed and compiled together with the OS kernel for real-time scheduling. In particular, the R3TOS supports not only traditional software tasks but also the software images of hardware tasks. The software image is obtained by extracting software features from hardware descriptions, such as task inputs and outputs. The software image gives the hardware tasks a software look-and-feel in the main CPU. Both software images and software tasks are integrated into the R3TOS and the whole system is then compiled into machine codes and stored in the program memory to be used by the main CPU. The main program starts with the software program in the host processor. When a hardware task is called by the main CPU, the task scheduler will add this task to the task queue, and it is then scheduled, allocated and executed by the R3TOS HWμK.

The communication between software and hardware is achieved by sharing IDBs and ODBs between the main CPU and the tasks. The tasks' IDBs and ODBs are virtually mapped into the processor's global memory. The task input is given by writing the IDB in the shared memory. After a task finishes execution, the result will be returned to the ODB and the CPU will be notified.

In addition, the R3TOS supports on-line task customization for application that supports folding mechanisms (see Chapter 6). In some applications, the number of the pipeline stage can be changed. Classic examples are database scanning or sequence alignment applications [Durbin1998, Oliver2005]. In such applications, tasks are most often composed of a sequence of pipelined PE arrays, which decides the task pipeline stage. Therefore the task can be folded by reusing the PEs to reduce task size [Isa2012]. In multi-user multi-task applications, the logic resources are dynamically and partially assigned to multiple users. In this context, fragmented resources may stay in an idle state if they are not big enough to accommodate a larger task, which reduces the efficiency of resource usage [Hong2013a]. In light of this, the R3TOS gives the ability to re-customize the size of tasks to make them fit into currently idle resources; whereby logic resources can be more efficiently used and overall system performance is improved. This on-line task customisability has been demonstrated in the context of a sequence alignment application [Hong2013b], as detailed in Chapter 6.

3.3 System Computing Model

The generated tasks are stored in the bitstream library, which is available to be called by the main CPU on-the-fly. Meanwhile, the task software and hardware features are extracted and stored in the main CPU and the HWμK respectively, which are then used to call, schedule and allocate hardware tasks during the system operation time. After a task finishes its computation, the resulting output is delivered to the next task's input using the proposed inter-task communication mechanism. If finished tasks do not need to be reactivated, they will be de-allocated and removed from the chip, so that the logic resources they used can be freed and reused for later tasks. The

following sections give overviews of task scheduling, task allocation and the inter-task communication mechanism.

3.3.1 Task Scheduling and Allocation

In the R3TOS, tasks are scheduled and allocated according to their time and area parameters, which are extracted during task generation. After task design and generation, the extracted task information includes a hardware version and a software version. The software version describes the task's software features, such as input arguments and returning results, which are stored in the main CPU memory and used in an identical manner as other software programs. The hardware version indicates the task's hardware behaviour, such as task time and area parameters, which are stored in the R3TOS HWμK to be used by the scheduler and allocator respectively. TABLE 3.1 gives an overview of the generated task parameters for both hardware and software versions. This chapter mainly focuses on the hardware features, which are the essential factors in hardware task scheduling and allocation. The hardware features can be classified into three types; namely, the basic information, time information and area information. Basic information about the task describes its current status and its task ID in the task queue. The time and area information is used by the task scheduler and task allocator respectively to schedule and allocate tasks during system operation time.

The basic task information includes the task ID and its current status. The task ID is the task identity in the task queue. The task queue is stored in the task memory, which includes all task information. In it, all tasks are queued and sorted according to their priorities, which depend on time constraints such as absolute deadlines. The parameter *Next_Task_ID* is a pointer to the address of the next task which has a lower priority. Compared with using linear memory space, this task queuing process can achieve better scheduling efficiency [Hong2011b]. Details of memory location are explained in Chapter 5.

TABLE 3.1 TASK PARAMETERS IN R3TOS

Hardware Features		Software Features
Task Basic Information	*Task_ID*	*Function_Name*
	Next_Task_ID	*Input_Arguments*
	Task_Status	*Input_Memory_Address*
Task Time Information	*Absolute_Deadline*	*Output_Result*
	Task_Arriving_Time	*Output_Memory_Address*
	Task_Allocating_Time	*Task_Status*
	Task_Execution_Time	
	Task_Finishing_Time	
Task Area Information	*Task_Width*	
	Task_Height	
	Task_Resource_Type	
	Task_Area	
	Task_Allocated_Position	

Task_status indicates the current operational condition of the hardware task, including if it is ready, executing, allocated, blocked, removed, or expired. Switching between different statuses is a fundamental mechanism of the R3TOS. Figure 3.5 shows a diagram of task status switching. In general, after a task is requested from the main CPU, it will be inserted into the task queue and scheduled according to its time constraint. The task with highest priority will be placed on the chip and starts executing immediately. If the currently available resources are not sufficient for the task, it will be blocked until another task is removed from the chip. Tasks that have finished their execution will stay in the allocated state, whereby they can be either reactivated by the host and restarted again directly, or removed from the chip to free up the logic resources for new tasks. Before beginning to place a task, the task deadline is firstly checked to ensure that it is not an obsolete task; otherwise the task is determined to be expired and the main CPU will be notified for post-processing.

In order to improve the evaluation of tasks in terms of both time and area, they are characterised in both time and area domains (see Figure 3.6). In the time domain (see Figure 3.6.a), a hardware task T_i can be described using both its absolute time features, such as arriving time r_i, allocating start time a_i, executing start time e_i, finishing time f_i, and deadline d_i; and their relative time features like allocating

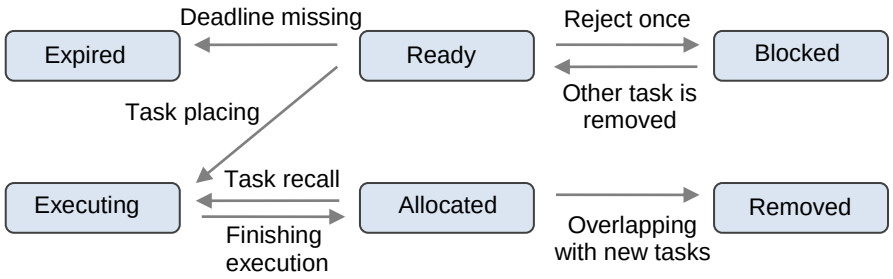


Figure 3.5 Task status switching

duration $T_{A,i}$, executing duration $T_{E,i}$, computing duration $T_{C,i}$, and relative finishing deadline D_i. Details of the timing information and its usage in scheduling algorithms is further explained in Chapter 4.

In the area domain (see Figure 3.6.b), a chip or a reconfigurable area can be modelled as a rectangle composed of $H_x \times H_y$ reconfigurable logic cells. Likewise, a task θ_i can be described as a $h_x \times h_y$ square, in which h_x and h_y are the number of reconfigurable logic cells in the horizontal and vertical directions respectively.

By using the above parameters, tasks in the R3TOS can be effectively measured in terms of both time and area, whereby the scheduling and allocating algorithms can be applied to schedule tasks in real time and manage hardware resources with high efficiency [Iturbe2013a]. To schedule tasks in real time, the Earliest Deadline First (EDF) algorithm is implemented, in which tasks are sorted according to their absolute deadlines, and the task with the nearest absolute deadline will be prioritized to be executed prior to others. To reduce the configuration overhead, non-preemptive scheduling is used, according to which tasks with status of executing cannot be interrupted, and hence complex hardware context switching is avoided. However, one potential disadvantage of the conventional EDF algorithm is that, if the task with the nearest deadline cannot fit into currently available resources, it will be blocked and other tasks with smaller sizes will be prioritised. In such cases, the blocked task will be more likely to expire if too many smaller tasks are prioritised or inserted. In the light of this, a novel EDF based algorithm is developed called the Finishing Aware Earliest Deadline First (FAEDF), which takes into consideration the currently

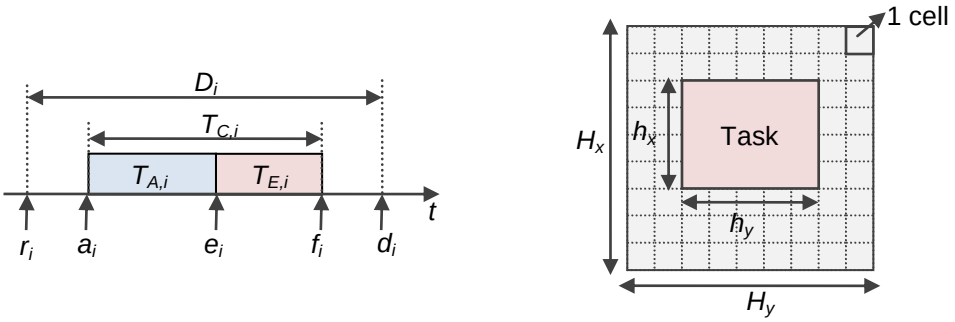


Figure 3.6 R3TOS hardware task modelling

executing task's finishing time in order to look ahead to see whether or not other smaller tasks should be inserted. In the FAEDF, when a task is blocked because resources are unavailable, other tasks are prioritised only if their insertion does not further delay the blocked task. By doing this, hardware tasks are more efficiently scheduled and more tasks can meet their deadlines [Hong2011b, Iturbe2013a, and Iturbe2013b].

On the other hand, in order to achieve better resource usage efficiency in the area domain, the task area information is used to improve the quality of placing. The quality of the task placement is measured by the compactness of the remaining logic resources. A highly compacted region contains larger intact free areas, where more tasks can be accommodated and resources are more efficiently used. To achieve this, two allocating algorithms are developed called the Empty Area Compaction (EAC), and the Empty Volume Compaction (EVC) algorithm [Hong2011a, Hong2011b, and Iturbe2013a]. In general, both of these algorithms evaluate placement quality by measuring the Maximum Empty Rectangle (MER). The MER is defined as the maximum number of adjacent empty CLBs in a rectangular area (see Figure 3.7).

When placing a task, the task allocator firstly finds out all positions that are possible to accommodate the task. Afterwards, the MERs remaining after placing the task are calculated for each position, and the position resulting in the maximum MERs is selected to allocate the task. Compared with other placement algorithms, these allocation algorithms achieve higher placement efficiency with a low implementation

footprint [Hong2011b]. The detailed algorithm design and test results for the scheduler and allocator are explained in the next chapter.

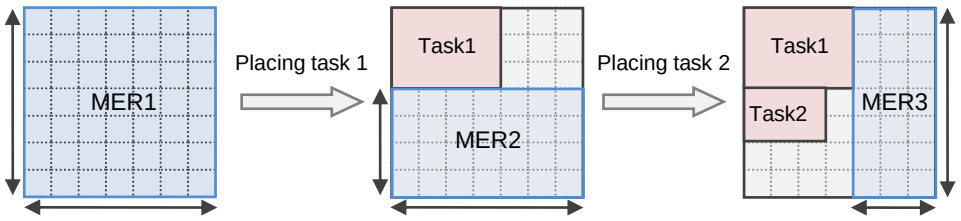


Figure 3.7 Maximum Empty Rectangles (MERs)

3.3.2 Inter-Task Communication

In the R3TOS, coupled tasks are arbitrarily allocated at separate positions, and therefore effective communication mechanisms have to be developed in order to make inter-task data exchange possible. To date, the R3TOS supports four different inter-task communication mechanisms used for different types of applications. These four mechanisms are: 1) Internal Configuration Access Port (ICAP) based communication; 2) Data Relocating Task (DRT) based communication; 3) shared memory based communication; and 4) the snake communication strategy (see Figure 3.8).

The ICAP based communication mechanism harnesses the on-chip configuration port and uses it for data communication [Iturbe2011b, Hong2013a]. Traditionally, the ICAP is used only for configuring the bitstream of predesigned static logic functionalities such as LUT configuration and for routing information [Iturbe2011b]. In effect, the bitstream also includes dynamic data information, such as memory data and register value. Therefore, by reading and writing the bitstream, the memory data can be obtained and changed respectively. As a result, the output of a task can be obtained by the main CPU through reading back the partial bitstream of the task's Output Data Buffer (ODB), and a task input can be given by reconfiguring the bitstream to update its Input Data Buffer (IDB). This ICAP communication mechanism establishes virtual channels between the task input/output buffers, which effectively eliminates traditional on-chip routing, and giving a route-less free

resource area for placing tasks arbitrarily (see Figure 3.8.a). The main advantage of this ICAP-based communication mechanism is its low area overhead, no communication routing resources are needed and all communication can be implemented by reusing already existing configuration hardware primitives. However the disadvantage of this approach is the low communication bandwidth, since the ICAP only supports 32-bit width at 100MHz. The throughput is further limited since the ICAP is not only used for data exchange, but also for configuring tasks and reading back the bitstream for fault diagnosis. For this reason, the ICAP-based communication is used for application tasks requiring less communication. Such tasks are also called computation domain tasks, or Low-Bandwidth Communication (LBC) tasks, which focus more on internal data computing rather than external data interaction. In contrast, communication domain tasks, or High-Bandwidth Communication (HBC) tasks are hardware modules requiring more external communication and data exchange. For such tasks, other communication mechanisms are developed to provide a higher bandwidth for inter-task communication, which will be described in the next section.

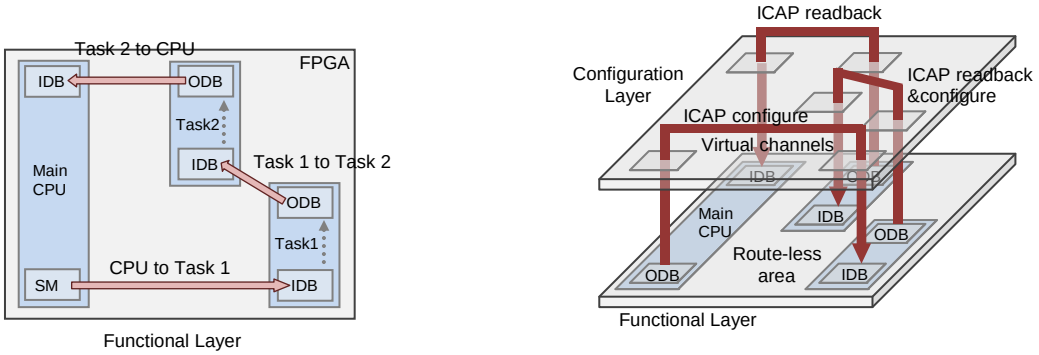
The communication mechanism based on the Data Relocating Task (DRT) creates a link between two tasks for direct data transmission [Iturbe2013b, Iturbe2013c]. The DRT includes the hardware configuration logic used to move data from a source BRAM-based task ODB to a destination BRAM-based task IDB using on-chip routing resources such as switch boxes (see Figure 3.8.b). The advantage of the DRT-based communication is that it allows a higher bandwidth for data exchange. In effect, the communication speed which can be achieved is as high as the maximum access speed of the BRAM, which has a 32-bit bandwidth running at about 100-200MHz, giving a throughput of between 0.4-0.8 GB/ s. Moreover, multiple DRTs can be allocated at the same time, which gives parallel communications between tasks and hence more tasks can communicate concurrently. However, the main disadvantage of the DRT is the area overhead, as the communication routings go across the region between two tasks, which severely fragments the chip area. Another limitation is that only BRAM-based task input/output buffers can be linked by DRTs. Given these factors, the DRT-based communication is suitable for

application tasks requiring high communication and data exchange, namely communication domain or HBC tasks.

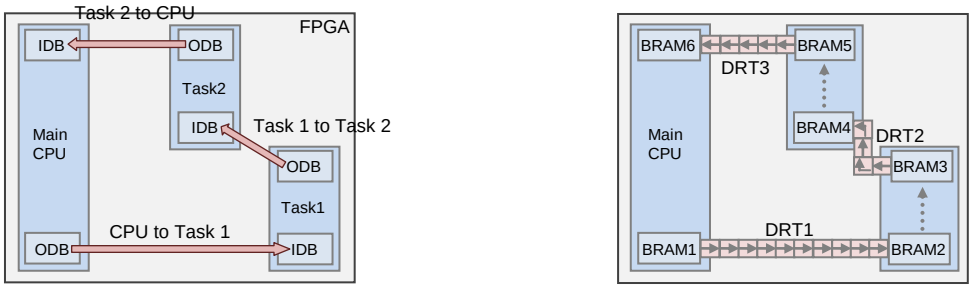
The communication based on shared memory is inspired by the Symmetric Multiprocessing (SMP) paradigm in multiprocessor programming. The R3TOS supports the virtual shared memory mechanism by synchronizing the content of physically separated BRAMs using the ICAP, whereby tasks in separate locations can share the same data and use it for communication and data exchange [Hong2012b]. Once a task changes the data in its shared memory, the host will be notified to readback the updated BRAM data and broadcast it to all other BRAMs (see Figure 3.8.c). In addition, the present system environment supports traditional SMP programming directives, including barriers, critical sections, and lock variables. Barriers give the ability to synchronise hardware tasks at any stage of a program's execution. Critical sections enable only one task to be executed exclusively. The locks are used by a task to prevent all other tasks from accessing particular data in the shared memory. Also, similar to the traditional SMP, all tasks have their own private memory used for internal computation. Moreover, to reduce the time overhead, Multiple Frame Writing (MFW) is used [Hong2012b]. MFW provides the ability to write identical data to multiple locations at an extremely high speed, which allows the updated BRAM data be broadcast to all other BRAMs in a very short time. Similar to ICAP-based communication, the shared memory mechanism gives a low area overhead at the expense of high data throughput, since no routing resource is used and all data is delivered through the configuration layer using the configuration port. This communication mechanism can be used for SMP-like applications, since it provides for compatibility with shared memory programming.

The snake strategy allocates tasks at places adjacent to the task it needs to communicate with, whereby tasks can communicate with each other locally using shared or adjacent BRAMs [Iturbe2011a]. This type of allocation tries to concatenate tasks by overlapping the consumer task's IDB BRAM with its producer task's ODB BRAM. In this task chain, previously finished tasks are removed from the tail and new tasks are added to the head, which is similar to the movement of a snake, and

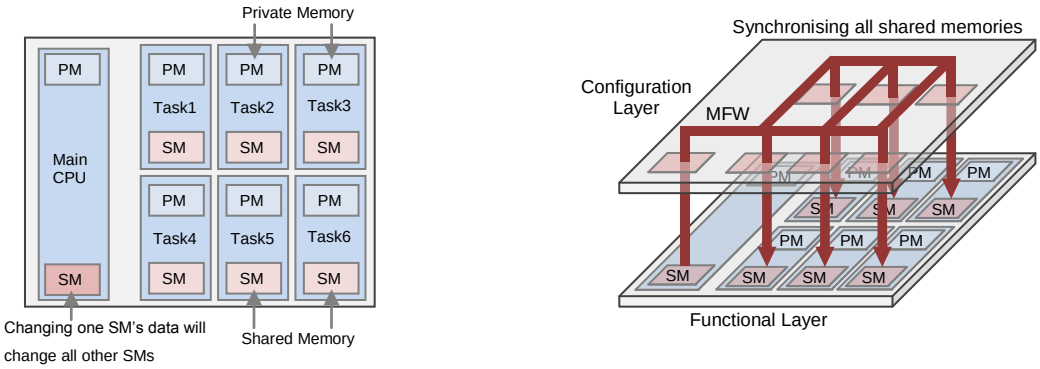
meanwhile the data is processed from the tail to the head (see Figure 3.8.d). This placement strategy gives high communication throughput as well as a low area overhead. Because the tasks are actually physically connected, there is no limitation on their communication bandwidth, and no communication routing is needed. The disadvantage of this approach is that tasks have to be particularly designed specifically to fit the chip architecture so that they can be concatenated, and thus placement flexibility and freedom are reduced. The snake strategy is suitable for data streaming applications, in which a high bandwidth can be provided for homogeneously structured processing elements.



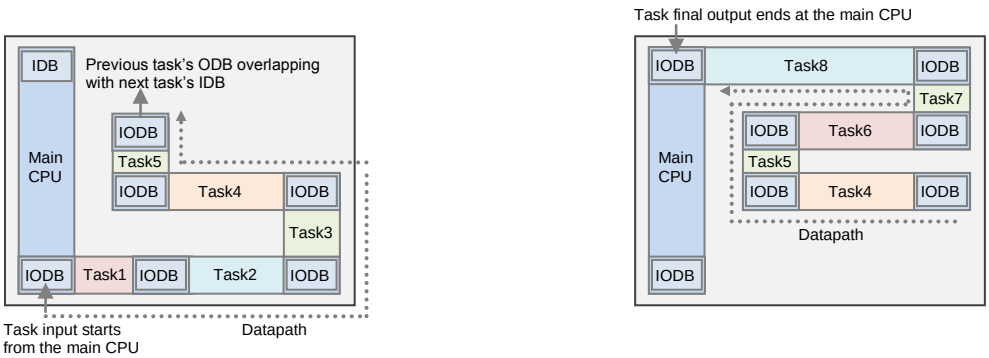
a) ICAP based communication



b) DRT based communication



c) Shared memory based communication



d) Snake strategy based communication

Figure 3.8 R3TOS inter-task communication mechanisms

3.4 Fault Tolerance

Presently, most FPGAs use SRAM-based technologies, which are susceptible to faults introduced by either intensive long-time internal overloads, such as Electro Migration (EM) and Time-Dependent Dielectric Breakdown (TDDB), or from external electromagnetic radiation, including Single Error Upset (SEU) and Hot Carrier Injection (HCI) [Iturbe2013c]. Emerging faults may provoke system failure if they are propagated to the system output (see Figure 3.9.a). The Mean Time To Manifest (MTTM) refers to the time duration needed for an internal error to be propagated the system output. In effect, some insignificant faults may not result in system failure, since they may be wiped by data overwriting. Xilinx Corp. has tested the failing rate for a typical application, and only 2% to 10% of total number of soft errors were found to result in system failure [Schumacher2012]. The Mean Time Between Errors (MTBE) is defined as the average time duration between the occurrence of two errors, which is measured as failures in time per billion hours (FIT). For example, the Rosetta Experiment showed that the Spartan-3 FPGA has an error rate of 111 FIT/Mb in CLBs, which gives an MTBE of 9.1 thousand hours in every million bits [Lesea2005]. Although the MTBE is an extremely long time period, it cannot be ignored since it decreases dramatically with an increase in the transistor density of a device and radiation intensity.

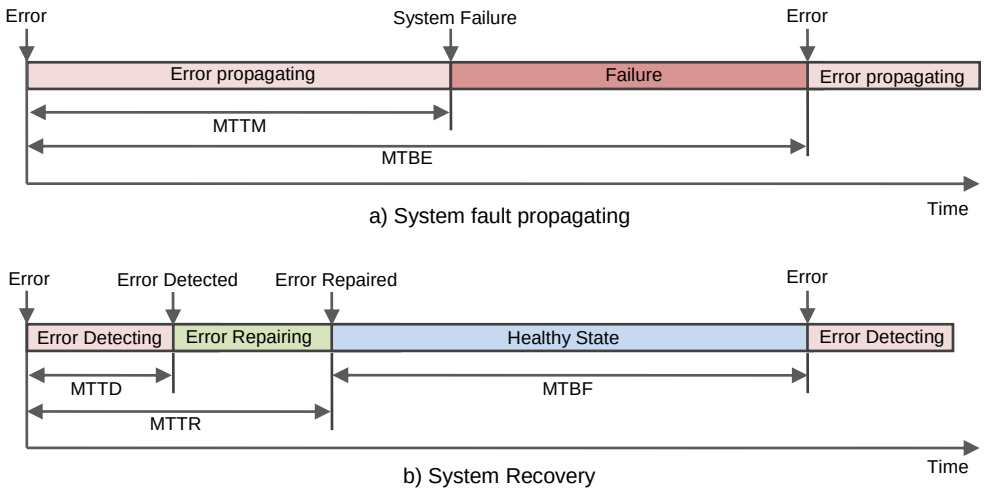


Figure 3.9 Time measurement for repairable system

In a fault tolerant system, emerging faults can be detected and repaired before the error is propagated to the system output, and the final system output thus remains correct. As depicted in Figure 3.9.b, the Mean Time To Detect (MTTD) is the time which elapses prior to detect the presence of an error, and the Mean Time To Repair (MTTR) is the total time needed to fix the error after it occurs. After an error is fixed, the error region returns to a healthy state, whereby the propagation of the error is prevented. The time interval between fixing the error and the next error emerging is called Mean Time Between Failure (MTBF), a term which is widely used in the tests of product reliability.

In respect of the above features, the R3TOS is intended to: 1) prolong or stop the propagation of faults by isolating tasks both physically and logically, thus increasing MTTM; 2) enhance fault detection ability by adding modular redundancies, thus improving the detection sensitivity; and 3) increase the speed of the detection of faults and to recover the system by applying existing on-chip facilities, thus reducing MTTD and MTTR. The following sections summarise the main methodologies used in the R3TOS to improve its fault tolerance in respect to the aforementioned three objectives.

3.4.1 Task Dependability and Fault Isolation

The most important role of a fault tolerant system is to detect and repair emerging faults, and to isolate faults from other non-faulty modules. To achieve this goal, modules or tasks in the system have to act as Fault Containment Units (FCUs). These ensure that faults are self-contained within each module, preventing them from propagating to other units [Lala1994, Kopetz2011]. In the R3TOS, all tasks are both physically and operationally independent FCUs, which self-contain regional faults within a single task without propagating them to other tasks. Physical isolation ensures that no hardware resource is shared between tasks, and operational isolation guarantees that during task operation there is no communication or data exchange between tasks. By doing this, all emerging faults can be contained within the module and isolated from others.

To physically isolate tasks, logic resources have to be exclusively used by a single task, whereby neither logic nor routing is shared with other tasks. Current synthesis tools such as Xilinx PlanAhead have provided the support to area-constrain a task within a square region. However, those generically used tools can only guarantee the exclusive usage of logic resources, such as LUTs, flip-flops and specific blocks, whereas routing resources cannot be strictly constrained within the task area. Unconstrained routings may go across other modules and cause short-circuits when shared with other tasks. To deal with this, Xilinx has recently released an Isolation Design Flow (IDF), which gives stricter constraints to separate task logic [Corbett2012], and provides Isolation Verification Tools (IVTs) to verify DPR designs.

Operational isolation ensures that an emerging fault is encapsulated in the task itself before it finishes its computing. After the computation finishes, the Cyclic Redundancy Check (CRC) calculates the bits of the output and its result is used to guarantee the reliability of the results. Therefore an emerging fault in the currently operating task can be detected before it is used for the next task. The CRC checking procedure can be applied to any of the four inter-task communication mechanisms, whereby faults can be isolated and contained in each task operation session.

3.4.2 Modular Redundancy

In the R3TOS, modular redundancy is provided for critical tasks, where a critical task can have multiple copies of instances to be executed at different positions at the same time. Similar to the Triple Modular Redundancy, which triplicates a task's functioning module and uses a majority voter to output a correct result if one of the three instances fails, the system triplicates reconfigurable regions; namely, the Fault Containment Computation Region (FCCR), so that the fault can be contained and isolated in one single FCCR.

Figure 3.10 gives an example of the redundancy model in the R3TOS. Here the reconfigurable region is divided into three partitions; namely, three FCCRs, in which multiple copies of tasks can be placed at different positions and can run in parallel,

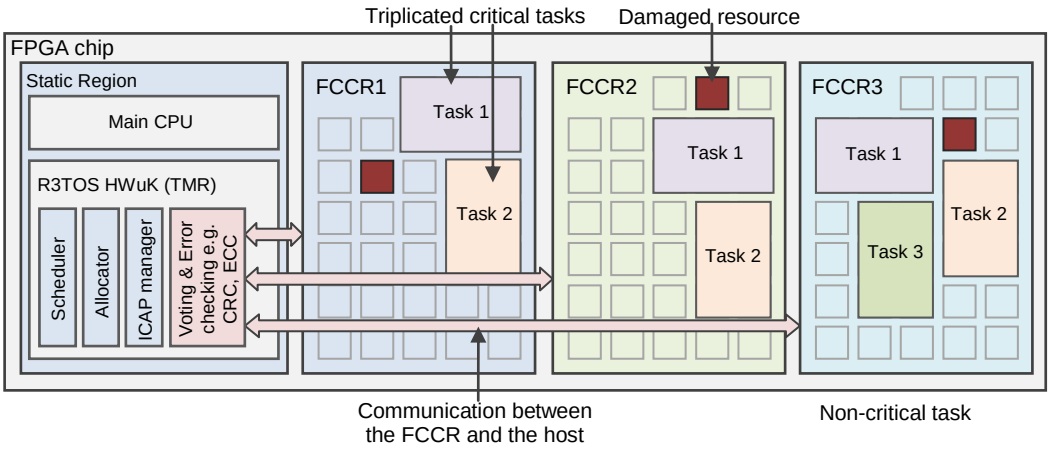


Figure 3.10 R3TOS redundancy model

allowing the damaged resource to be avoided. The critical tasks, task1 and task2, are triplicated (TMRed) and copied in each FCCR. The less-critical tasks can have less than three copies, e.g. either two or one, such as task3. After a task finishes its computation, the output result is returned to the host, and the post-checking is performed for redundant modules. For triplicated tasks, if one instance fails because of a transient soft error, the majority voter will select the correct result produced by the other two instances. It is assumed that only one instance incurs a fault during task operation, based on the fact that in reality the task execution time is significantly shorter than the fault rate. To accelerate the voting process, a 32-bit CRC module is implemented to compute the CRC code for the output result when it is readback through the ICAP. Thereby to check for result correctness, it is only necessary to compare the CRC code instead of a large amount of data. Once a faulty module is detected, the whole module is reconfigured to erase transient soft errors such as Single Error Upsets (SEUs). For double modular redundancy, both modules will be reconfigured and restarted if their resulting CRC codes are different. If the fault still persists after reconfiguration, it will be marked as a damaged resource, which is a permanent fault that cannot be fixed and thus should be avoided.

To reduce the configuration time overhead spent on replicating multiple instances, the Multiple Frame Writing (MFW) is used to rapidly replicate multiple copies of instances at different positions. The MFW is realised by changing the bitstream format from the one-data-one-address writing mode to the one-data-multi-address

writing mode. MFW significantly improves the configuration speed by at least 2x. The detailed implementation of this is elaborated in Chapter 6.

While the reconfigurable region is protected by modular redundancy, the static region is also protected using the more reliable TMR, which triplicates all the system kernels in the HWμK to provide the R3TOS with full protection. It is acknowledged that some non-redundant components, such as the voter and the physical ICAP port, cannot have multiple instances. In light of this limitation, additional fault detection and recovery mechanisms are developed, such as periodic diagnoses and ECC protection, which are introduced next.

3.4.3 Fault Detection and Recovery

In addition to modular redundancy, the R3TOS enhances its fault tolerance by harnessing its fault detection and recovery mechanisms to cope with different types of errors. Central to this, a fault tolerant soft microprocessor is designed and implemented by enhancing the existing ECC memory, whereby a single fault can be automatically fixed and double faults can be detected for all the components in the HWμK. Moreover, the scrubbing technique is applied to periodically diagnose the system functionality at runtime.

ECC protected microprocessor

The R3TOS HWμK consists of three components; namely, the task scheduler, the task allocator, and the ICAP manager. These are separately implemented on three softcore PicoBlaze microprocessors. Normally, the PicoBlaze microprocessor uses regular BRAMs for both program memory and data memory, which is highly susceptible to radiation-induced faults. In effect, because of the high integration and high transistor density of BRAMs, they suffer an error rate at about 2-4 times higher than other components [Lesea2005]. To counter this threat, Xilinx has developed a built-in ECC circuitry for their BRAMs, whereby one single bit error can be automatically corrected and double errors can be detected [Xilinx2008, Xilinx2012c]. However, this Error-Correcting Code (ECC) mechanism cannot be applied to the program memory of the PicoBlaze processor because of the

synchronisation problem. For instance, the ECC-BRAM needs two clock cycles to read data, whereas the operation of the PicoBlaze microprocessor assumes memories with one clock cycle latency. To circumvent this problem, a hardware interface is developed called the ECC Processor Adaptor (EPA), which solves the synchronisation problem by "looking ahead" and pre-fetching the next instruction. In addition, recovery hardware mechanisms are also developed to automatically correct a single error without interrupting system operation. Here the fault is fixed at the back-end of the system, thus having no effect on front-end system performance.

While the processor memories are protected by ECC-BRAM, the fault tolerance of the processor logic is also enhanced by giving triple redundancy for all of the processor hardware cores. In all of the HWμK components, including the task scheduler, the task allocator and the ICAP manager, both PicoBlaze kernels and the EPA are triplicated and two voters are used to interface with the peripheral and instruction buses respectively (see Figure 3.11). The detailed implementation of the EPA and the recovery hardware mechanism is presented in Chapter 5. Compared with previous similar work, which triplicates both processors and BRAMs [Heiner2008]. The proposed ECC-protected processor not only saves large memory resources, but also provides the ability to automatically recover from a transient fault.

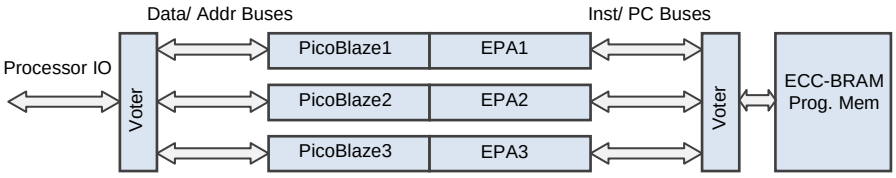


Figure 3.11 ECC protected microprocessor

Scrubbing technique

The scrubbing technique is used to dynamically readback the bitstream from the configuration memory in order to check its correctness at runtime. The bitstream includes the configuration information for all the logic resources, such as the content

of LUTs, routing switches and flip-flop settings, which record all the functionalities of an FPGA. Besides this, the bitstream also contains temporary data information such as memory contents and register values in flip-flops. When performing system scrubbing, the temporary data information is masked out, and therefore only the information of logic functionalities is checked. There are three ways to check the correctness of a bitstream. Firstly, the readback bitstream can be compared with the original bitstream stored in a non-volatile memory, which is sometime called as "golden bitstream" and is assumed to be correct at all times. Alternatively, the correctness of the readback bitstream can be checked using the FrameECC, which is an existing built-in hardware primitive that automatically corrects a single bit error and detects double bit errors when a frame is readback through the ICAP. In addition, "blind scrubbing" refers to directly rewriting the configuration memory using the golden bitstream, whereby the readback step is omitted and the configuration memory is more frequently refreshed to wipe out upcoming faults. In the R3TOS, the FrameECC mechanism is chosen to be implemented since it provides dynamic diagnosis without requiring extra memory space.

Figure 3.12 gives a basic flow diagram of system diagnosis. During system operation, a fault can be detected from either a modular redundancy mismatch or by the periodically performed scrubbing. If a fault is detected in the task reconfigurable region, the faulty task will be reconfigured, whereby the software transient error in the configuration memory can be overwritten. After the task is restarted, a second readback is performed to confirm that the fault has been corrected by reconfiguring. If the fault still persists, the faulty resource will be marked as a damaged resource which cannot be fixed and thus should be avoided. If the fault is detected in the system static region, a full reconfiguration of the whole chip will be performed. Since the reconfiguration mechanism itself is not trust-worthy and therefore the whole system needs to be reset to its initial stage. In the worst case, if the fault in the static region cannot be fixed by full reconfiguration, the system will be shut down for safety protection.

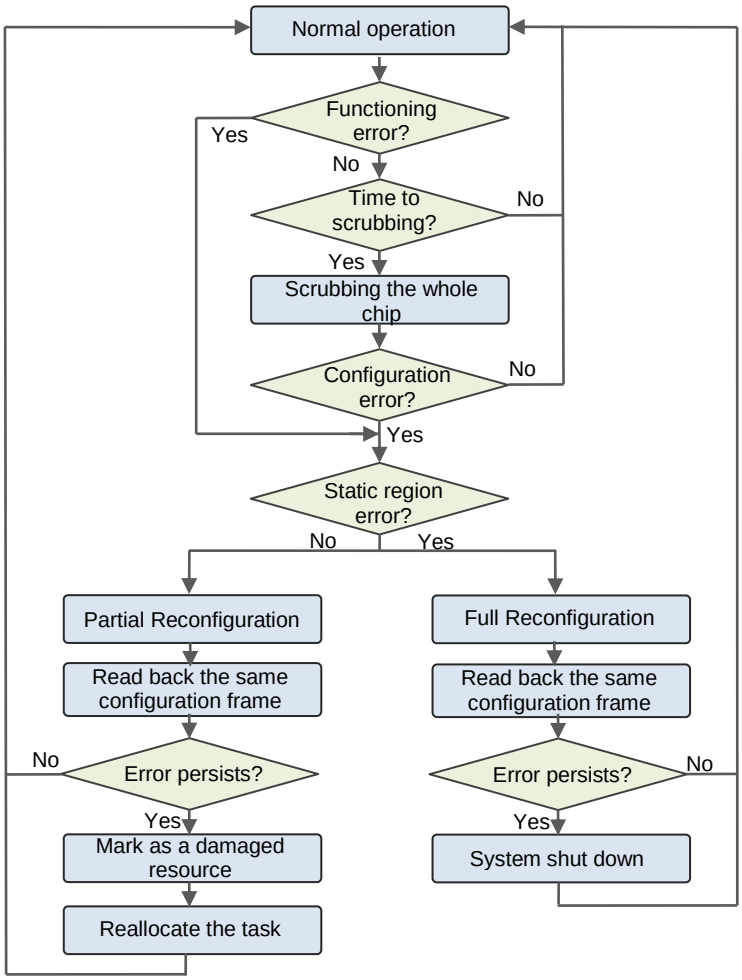


Figure 3.12 Basic system diagnosis

3.5 Conclusion

This chapter briefly introduced the overall operational model of the R3TOS, from the task design phase, via the task operating phase, to the fault tolerance phase. The basic concept of the R3TOS is to efficiently manage reconfigurable hardware resources in the background and to provide application designers with a reconfigure platform where hardware tasks behave just like software tasks.

In the task design phase, the user needs to co-design their application program in either software or hardware to be executed in the main CPU and FPGA hardware respectively. The FPGA hardware is then partitioned into less-coupled tasks, and

wrapped with TCLs to generate task bitstreams, which are stored in the bitstream library to be called during system operation. In addition, software images are created for all of the hardware tasks to describe their software features, which are then registered in the main CPU to be used by the main program. During the system operating phase, tasks are scheduled in real-time and placed at optimised positions by the task scheduler and allocator respectively, using the proposed novel efficacious scheduling and allocating algorithms. Inter-task communication can be performed using one of four communication mechanisms according to the requirements of the application task, such as HBC or LBC. When an error occurs during operation, the system can give quick responses to detect, isolate, and fix faults before they are propagated to other modules. This is achieved by physical and logical task isolation, modular redundancy, and a variety of fault-proofing techniques, such as ECC protection, CRC checking and scrubbing.

Benefiting from the above mechanisms, the R3TOS significantly improves the efficiency of the usage of the reconfigurable hardware resources, whereby both real-time performance and operational reliability are improved. The subsequent chapters give more detailed information on the design and implementation of the R3TOS system. Chapter 4 describes the high-level methodologies and algorithms used to design the R3TOS, and Chapter 5 elaborates on the detailed hardware implementation of the system.

Chapter 4

Design of Algorithms for the Scheduling and Allocation of Tasks

The R3TOS supports efficient runtime scheduling and allocation of hardware tasks. In traditional software-based real-time operating systems, a task scheduler is used to schedule tasks in real-time to meet their time constraints, and a task allocator aims to allocate tasks in optimised positions to reduce memory fragmentation. Likewise, in the R3TOS, a task scheduler and a task allocator are designed to schedule hardware tasks in real-time and to allocate them to hardware resources. The main difference between the software and hardware in scheduling and allocation is that the software task is executed in sequence and allocated in one-dimensional memory space, whereas hardware tasks can be executed in parallel and mapped to two-dimensional hardware logic resources. Due to this difference, hardware scheduling and allocation become more challenging not only because multi-executing tasks increases scheduling complexity, but also since the existence of heterogeneous resources adds to the difficulty of allocation. To circumvent this, a novel real-time scheduling algorithm is developed called the Finishing Aware Earliest Deadline First (FAEDF) algorithm, which extends the Earliest Deadline First (EDF) by looking ahead to task finishing times [Hong2011b, Iturbe2013a]. The scheduling algorithms are used together with two novel and efficacious allocation

Related publications: [Hong2011a], [Hong2011b], and [Iturbe2013a].

algorithms; namely, the Empty Area Compaction (EAC) algorithm and the Empty Volume Compaction (EVC) algorithm [Hong2011a, Hong2011b and Iturbe2013a]. The scheduling and allocation algorithms are used together to analyse and optimise the 3-D computing space, where the scheduling algorithm schedules tasks in a 1-D time domain, and the allocation algorithm allocates tasks in the 2-D area domain. In effect, scheduling and allocation are not strictly separated in time or area, but interact with each other to achieve better performance, so that the scheduler uses area information when scheduling tasks and the allocator allocates task in consideration of both time and area information. The following two sections illustrate the scheduling and allocation algorithms, elaborating on their functioning and coding and evaluating their simulation results and comparing them with other approaches. The work in this chapter was developed in cooperation with the author's colleague, Xabier Iturbe, and some of the ideas and results have been published [Hong2011a, Hong2011b, and Iturbe2013a].

4.1 Task Scheduling Algorithms

To schedule hardware tasks in real-time, the FAEDF algorithm is developed and compared with the widely used EDF algorithm. Compared with the traditional EDF algorithm, in which tasks with the nearest deadline are prioritised, the FAEDF algorithm also takes into consideration task finishing time, so that more tasks can meet their time constraints. Before introducing the details of the computation and coding for both algorithms, the timing parameters that will be used in the scheduling algorithms are first introduced.

Figure 4.1 and TABLE 4.1 give the detailed timing figures used to schedule a task. A hardware task T_i has its *computing duration* $T_{C,i}$, which is the sum of the *allocating duration* $T_{A,i}$ and the *executing duration* $T_{E,i}$. The *allocating duration* $T_{A,i}$ is the time duration for configuring a task to the chip, which depends on the task area (or the bitstream size) and the throughput of the configuration port. The task *executing duration* $T_{E,i}$ is the product of multiplying the clock cycles required by the task by the system clock period. The task's *relative finishing deadline* D_i refers to the maximum

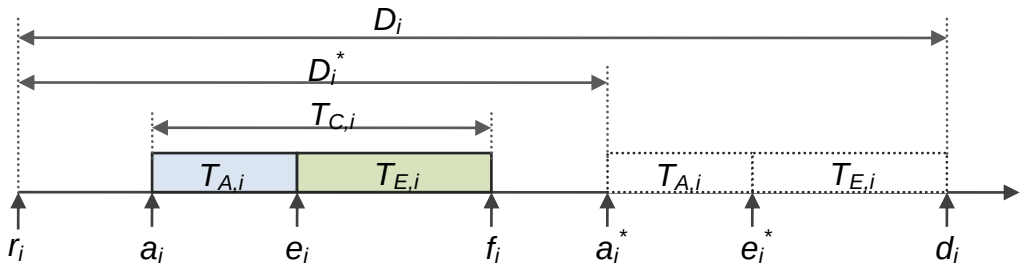


Figure 4.1 Task timing parameters used for scheduling

TABLE 4.1 TASK TIMING PARAMETERS USED FOR SCHEDULING

symbol	Parameter name	Explanation
$T_{A,i}$	*Allocating duration*	Elapsed time to allocate a task
$T_{E,i}$	*Execution duration*	Elapsed time to execute a task
$T_{C,i}$	*Computing duration*	Elapsed time to allocate and execute a task
D_i	*Relative finishing deadline*	maximum relative affordable elapsing time before a task's completion
D_i^*	*Relative allocating deadline*	maximum relative affordable elapsing time before a task starts allocating
r_i	*Absolute arriving time*	the time a task is released
a_i	*Absolute allocating start time*	the time a task starts allocating
e_i	*Absolute executing start time*	The time a task starts executing
f_i	*Absolute finishing time*	The time a task finishes executing
a_i^*	*Absolute latest allocating time*	The latest affordable time to allocate a task
e_i^*	*Absolute latest executing time*	The latest affordable time to execute a task
d_i	*Absolute finishing deadline*	The latest affordable time to finish a task

affordable time which can elapse before the task's completion, and the *relative allocating deadline* D_i^* refers to the maximum affordable time before T_i starts its allocation. By giving a task an *executing duration* $T_{E,i}$, *allocating duration* $T_{A,i}$, and *relative finishing deadline* D_i, the task *computing duration* $T_{C,i}$ and *relative allocating deadline* D_i^* can be calculated using Equation 4.1.

$$\begin{cases} C_i = T_{E,i} + T_{A,i} \\ D_i^* = D_i - T_{E,i} - T_{A,i} \end{cases}$$

Equation 4-1 Task relative time parameter deduction

The task's absolute time parameters can be obtained if its *arriving time* r_i and *allocating start time* a_i are given. The task's *absolute finishing deadline* d_i is the latest affordable time a task can be completed, and the task's *absolute latest executing time* e_i^* is the absolute deadline to start executing a task. Assuming that a task arrives in an *arriving time* r_i, and *allocating start time* a_i, the other absolute parameters can be obtained from Equation4.2:

$$\begin{cases} e_i = a_i + T_{E,i} \\ f_i = a_i + T_{E,i} + T_{A,i} \\ a_i^* = D_i + r_i - T_{E,i} - T_{A,i} \\ e_i^* = D_i + r_i - T_{E,i} \\ d_i = D_i + r_i \end{cases}$$

Equation 4-2 Absolute time parameter

Based on the above time parameters, hardware tasks can be effectively scheduled using the proposed real-time scheduling algorithms, namely non-preemptive EDF and FAEDF. The following sections present the functioning and coding for these two algorithms as well as their simulation results.

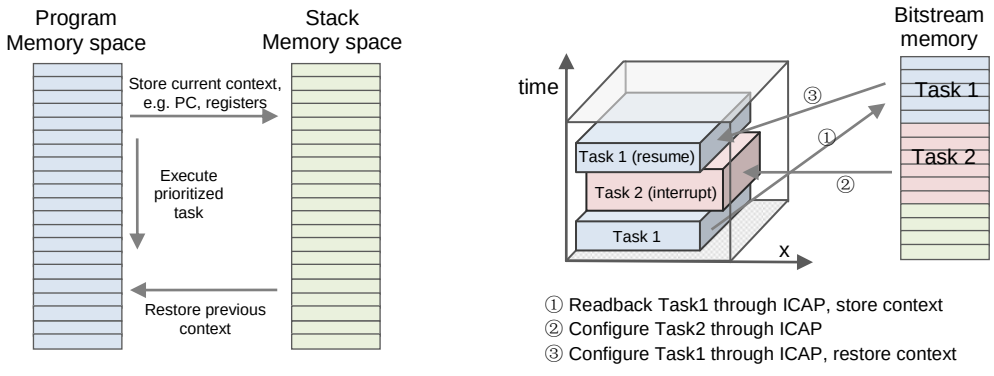
4.1.1 Non-preemptive Early Deadline First (EDF) Algorithm

The EDF algorithm has been widely used for scheduling real-time tasks the mono-processor applications, due to its simplicity of coding and practicability of execution [Danne2005, Dittmann2007]. Likewise, hardware tasks can be scheduled according to the same principle, since the ICAP is accessed both exclusively and sequentially, which is similar to sequential task scheduling in the single-processor system. The EDF algorithm prioritise the task with the nearest deadline, which could be either the task relative finishing deadline D_i as is used in Deadline Monotonic (DM) systems [Leung1982] or the task absolute finishing deadline d_i [Liu1973]. The latter gives more realistic scheduling, since it can dynamically update the scheduling scheme

according to the arrival times of tasks. Therefore the absolute deadline is used in R3TOS scheduling.

One difference between software and hardware scheduling is that hardware requires more scheduling overheads in either setting up a task to the hardware resources, or preempting a task from the hardware. In order to set up a task, complicated processes need to be performed, including searching for an optimised location, modifying the bitstream, and loading the bitstream to the configuration memory through the ICAP, which is more time-consuming than the software task creation. Similarly, to preempt a hardware task, all of the current register values, as well as the memory data, need to be stored for context switching, which requires reading back bitstreams from the configuration memory, and writing stored bitstreams back again to restore the context (see Figure 4.2).

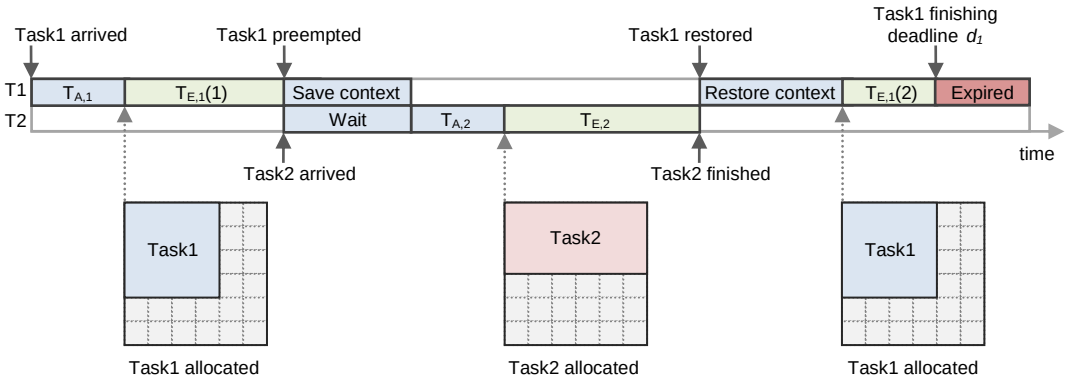
Although hardware context switching has been proven to be possible [Simmler2000, Ahmadinia2004a, and Jovanovic2007], the R3TOS uses non-preemptive EDF in order to avoid the large overhead associated with context switching. In effect, due to the large time overhead involved in preempting a task, more tasks will be delayed and miss their deadlines. For example, Figure 4.3 shows a situation where tasks can be better scheduled using non-preemptive EDF. In Figure 4.3.a, *task1* (*T1*) misses its finishing deadline d_1 since it is delayed by context switching, which induces a larger time overhead, whereas in Figure 4.3.b both tasks can finish their computation before their finishing deadlines.



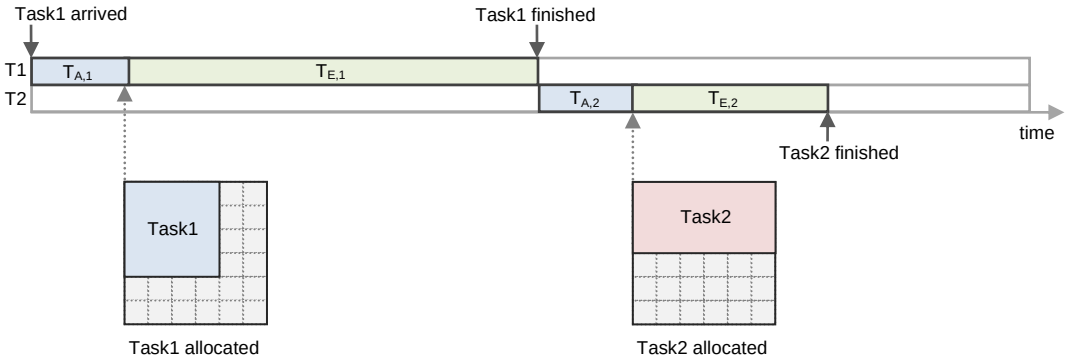
a) Software task preempt

a) Hardware task preempt

Figure 4.2 Preempt tasks in hardware and software



a) Preemptive EDF scheduling, task1 expired



b) Non-preemptive EDF scheduling, both tasks finished

Figure 4.3 Preemptive vs. Non-preemptive EDF algorithm

Note that in this example the main reason that *T1* misses its deadline is the large overhead of context switching, which includes reading back the bitstream, allocating a new task and delivering the task's input data. Therefore, to take into account the setting up overhead, the *Absolute latest allocating time* (a_i^*) is used instead of the absolute finishing deadline to schedule tasks. The a_i^* is the latest time to start configuring (set up) the task to meet its deadline. In the task queue, all tasks are sorted in order of increasing values of a_i^*. Algorithm 4.1 gives the pseudo-code for the non-preemptive EDF algorithm. The algorithm takes the task queue *R* (sorted by increasing a_i^*) and the maximum empty rectangle of the chip (*Chip-MER*) as its inputs, and outputs the prioritised task after scheduling. Here, tasks in the task queue are sequentially scanned and compared with the *Chip-MER* (lines 2 and 3). A size

smaller than the Chip-MER means that the task's area is smaller than the biggest area of the resources available, and thus the task is passed to the allocator since it is possible to be allocated (line 4). However, given a smaller area it cannot be guaranteed that the task shape, in terms of task width and height and resource type, can fit into the available resource. If the task cannot be allocated, the allocator will return with *null* (line 4) and the next task in the task queue will be scheduled. The algorithm finishes and returns to the task pointer if the currently scheduled task can be allocated (line 5); otherwise, it returns to *null* (line 6).

Inputs:

1. Ready task queue R sorted by increasing a_i^*
2. Chip-MER from the allocator

Outputs:

Prioritized task

```
1.  begin:
2.    for( i = 1, i< R, i++ )
3.      if (h_{x,i} × h_{y,i} ≤ MER)
4.        if (allocate(i) ≠ null)
5.          return i;
6.    return null;
7.  end
```

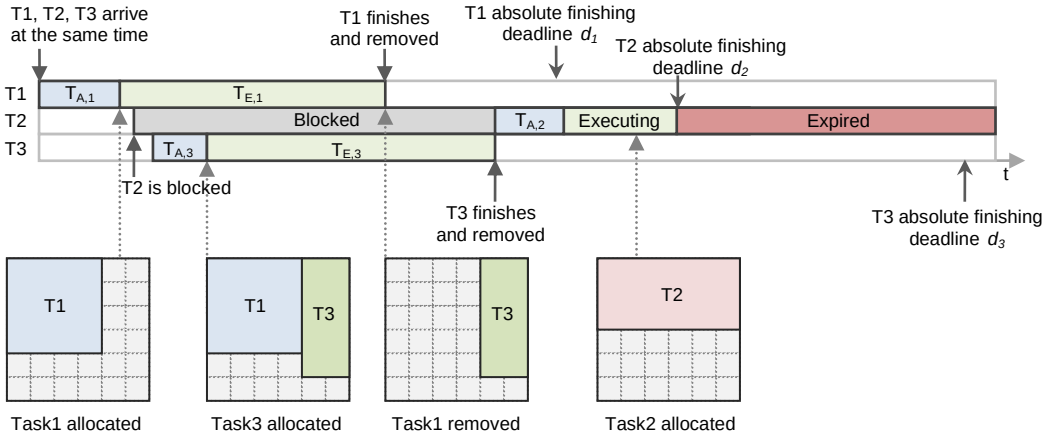
Algorithm 4.1 Non-preemptive EDF scheduling algorithm

The non-preemptive EDF is one of the simplest scheduling algorithms, which requires less computing time and gives satisfactory efficiency [Hong2011b]. However, if we are not trying to achieve the fastest scheduling speed, a better scheduling efficiency can be obtained by using the FAEDF algorithm.

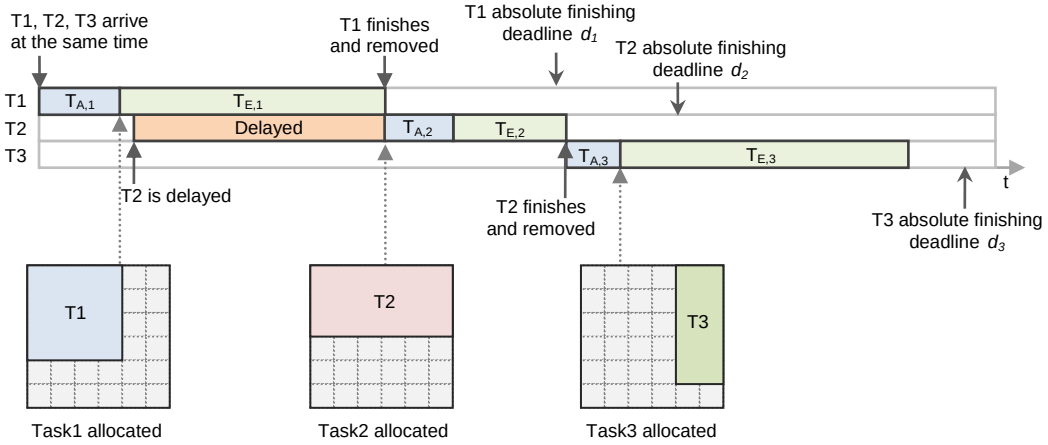
4.1.2 Finishing Aware Earliest Deadline First (FAEDF) Algorithm

In general, the FAEDF algorithm is an extension of the traditional EDF algorithm, and their functions are similar except that the FAEDF gives extra awareness of the task finishing time. Although the FAEDF requires extra computing time, it achieves better task finishing rates by looking ahead at the finishing time of tasks.

The FAEDF algorithm takes into account finishing times of other tasks before blocking the currently scheduled task. In the traditional EDF, if a task cannot be allocated, it will be blocked to allow the next task to be scheduled. In contrast, the FAEDF does not block the current task, but postpones its allocation to a later time that still meets the task's deadline. The duration of postponing time is decided according to tasks which are already in their executing stage but will soon be finished and de-allocated. In other words, the allocation of a currently scheduled task can be delayed to a later time, at which some other tasks will be removed from the chip, leaving more free area for this current task. Figure 4.4 gives an example where more tasks can be scheduled when using the FAEDF algorithm. In this example, three tasks arrive at the same time with different absolute finishing deadlines ($d_1 < d_2 < d_3$), which are to be scheduled by non-preemptive EDF (Figure 4.4.a) and FAEDF (Figure 4.4.b). In both cases, *task1* (*T1*) is allocated first since it has the nearest deadline d_1, and *task2* (*T2*) is going to be allocated after *T1*. However, since there is not enough resource space after *T1* is placed, *T2* cannot be immediately placed. In such a situation, the non-preemptive EDF will block *T2* and schedule the next task *T3* (see Figure 4.4.a). Consequently, the insertion of *T3* further delays the execution of *T2*, and *T2* has to start after both *T1* and *T3* are removed. As a result, *T2* misses its finishing deadline d_2. In contrast, with the FAEDF algorithm (see Figure 4.4.b), when *T2* cannot be immediately placed, it will be delayed, since the algorithm is aware both that 1) by the time *T1* finishes its computation, *T2* can still meet its *absolute latest allocating time* a_i^* (which represents time awareness), and 2) *T2* can be placed after *T1* since its size is larger than *T2* (which indicates area awareness). Therefore, the allocation of *T2* is delayed rather than blocked, and *T2* can be placed as soon as *T1* finishes its computation. As a result, all tasks are scheduled in line with their deadlines and all of them meet their finishing deadlines.



a) Non-preemptive EDF scheduling, task2 expires



a) FAEDF scheduling, all tasks finish computing

Figure 4.4 Non-preemptive EDF algorithm vs. FAEDF algorithm

On the other hand, to further improve the efficiency of resource usage, during the delay period, if other tasks with a lower priority can be completely allocated before the *absolute latest allocating time* (a_i^*) of the delayed task, that is, $e_{inserted\ task} < a_{delayed\ task}^*$, then the other task can be inserted into the delay period. For example, as shown in Figure 4.5, *task3* (*T3*) is inserted during the delay period of *task2* (*T2*), for the reason that *T3* can be completely allocated before the *absolute latest allocating time* of *T2* (a_2^*), i.e. $e_3 < a_2^*$. In this example, although the insertion of *T3* further

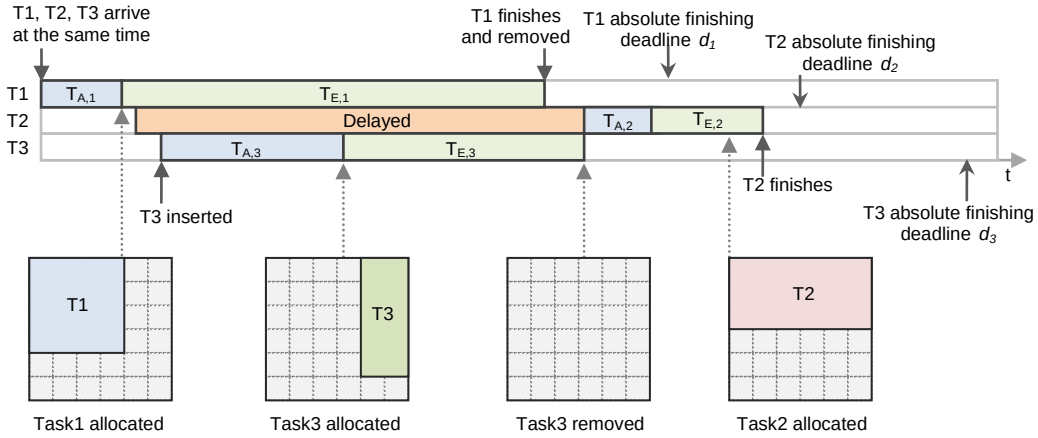


Figure 4.5 FAEDF scheduling, task3 insertion

delays *T2's* execution, *T2* can still meet its deadline. Note that the allocation time ($T_{A,i}$) is taken into consideration, rather than the execution time ($T_{E,i}$) or the whole computing time ($T_{C,i}$), based on the fact that allocation is the main critical path in task scheduling. This is due to the exclusive access to the ICAP, where tasks can be executed in parallel but cannot be allocated in parallel. In addition, all the executing tasks are sorted by their absolute finishing time f_i. Hence the task with the nearest finishing time will be scanned first, which reduces the possibility of further delaying the current task.

Moreover, the FAEDF will be activated only if the *average time constraint* (C_t) is loose; otherwise it behaves just like the non-preemptive EDF algorithm. C_t is the sum of every task's *allocation duration* $T_{A,i}$ divided by its *relative allocation deadline* D_i^*. C_t is defined by Equation 4-3, in which $T_{A,i}$ is the *allocation duration*, D_i^* is the *relative allocation deadline*, and *R* stands for the *ready task queue*.

$$C_t = \sum_{i=1}^{i=R} \frac{T_{A,i}}{D_i^*}$$

Equation 4-3 Average time constraint

C_t implies that the overall deadline tightness of all tasks, that is, C_t is high when the allocation duration of tasks is close to their relative allocation deadlines; and a low value of C_t indicates that tasks have more time redundancy. In the extreme case, C_t approaches its highest value *R*, when all tasks have the same $T_{A,i}$ and D_i^*, which

means that all tasks have to be allocated as soon as they arrive. C_t will approach zero if the tasks' D_i^* are much higher than their $T_{A,i}$, in which case all tasks can be scheduled easily and flexibly to meet their deadlines. In the proposed FAEDF algorithm, C_t is used to decide whether the currently scheduled task should be delayed or blocked. If C_t is below a certain threshold, which means that the time constraint is not high, the task will be delayed and other tasks can be inserted. Otherwise, if C_t is above the threshold, the algorithm behaves just like the normal non-preemptive EDF, which does not enable task delay and insertion. The reason for using a threshold is that, if the time constraint is high, there is no point in trying to delay tasks since they will be more likely to miss their deadlines. In other words, FAEDF only improves the quality of scheduling tasks if the deadlines of tasks are loose. C_t is updated every time a task is scheduled, which gives the algorithm the capacity for dynamic adjustment to current task constraints.

Algorithm 4.2 gives the pseudo-code of the FAEDF algorithm. The algorithm has four inputs, including: the ready task queue *R*, the executing task queue *E*, the *Chip-MER* from the allocator, and the updated average time constraint C_t. The ready task queue contains all the arriving tasks ready to be allocated, which are sorted by their *absolute latest allocating time* a_i^*. In addition, the executing task queue contains all the existing placed tasks sorted by their *absolute finishing times*, which are used to look ahead to see if any task will finish before the allocating deadline (a_i) of the currently scheduled task. The FAEDF behaves like non-preemptive EDF if $C_t <$ *Threshold* (lines 2 to 5), and simply allocates the task with the nearest allocating deadline (a_i^*) if its size can fit into the currently available resources. The FAEDF extension section will be enabled only if the current *average time constraint* is below the threshold, i.e. $C_t <$ *Threshold* (lines 6 to 15). In the extension section, the algorithm firstly scans all of the executing tasks (line 7), and if there is any task T_j which can finish its execution before the *absolute latest allocating time* (a_i^*) of the current task T_i (line 8), the *insertion* flag is set to one (line 9) and the current task T_i will be delayed rather than blocked (line 10). Furthermore, if the *insertion* flag is one (line 12), the other ready tasks in the *ready task queue R* will be scanned (line 13). If a task can finish its allocation before the *absolute latest allocating time* (a_i^*) of the

current task T_i, i.e. $t_{current} + T_{A,m} < a_i^*$ (line 14), then T_m can be inserted (line 15). The function will return to the currently scheduled task T_i (line 5), if T_i can be allocated. Otherwise it will return to the inserted task T_m (line 16), if T_m is inserted and allocated by the allocator. If neither of them is allocated, the function returns to *null* (line 19).

Inputs:

1. Ready task queue R sorted by increasing a_i^*
2. Chip-MER from the allocator
3. Executing task queue E, sorted by increasing f_i
4. Average time constraints C_t

Outputs:

Prioritized task

```
1.  begin:
2.    for( i = 1, i< R, i++ ) {
3.      if ( h_{x,i} × h_{y,i} ⩽ MER )
4.        if ( allocate(i) ≠ null )
5.          return i;
6.      if ( C_t < Threshold ) {
7.        for( j = 1, j < E, j++ )
8.          if ( f_j < a_i^* ) {
9.            Insertion = 1;
10.           delay T_i ;
11.         }
12.       if ( Insertion == 1)
13.         for( m = 1, m < R, m++ )
14.           if ( t_current + T_A,m < a_i^* )
15.             if ( allocate(m) ≠ null )
16.               return m;
17.     }
18.   }
19.   return null;
20. end
```

Algorithm 4.2 FAEDF scheduling algorithm

4.1.3 Test Results of the Scheduling Algorithms

Simulation is used to compare non-preemptive EDF and FAEDF, in respect of task finishing rate and the algorithm's execution time. The task finishing rate is the percentage of successfully scheduled tasks that meet their finishing deadlines, and the algorithm execution time is the time spent on making a scheduling decision for one task. These two features are tested when two different conditions are varied, namely; the number of damaged resources, and the *average time-area constraint* $C_{t,a}$.

The *average time-area constraint* $C_{t,a}$ is the average strictness of task deadline requirements taking into consideration both time and area, and it is used to set up an benchmark to measure the quality of scheduling algorithms. The *average time-area constraint* $C_{t,a}$ is defined in Equation 4-4, in which H_x and H_y are the chip width and height respectively; h_x and h_y are the task width and height; $T_{C,i}$ is the duration of task computation, D_i is the relative finishing deadline; and R is the total number of tasks in the task queue.

$$C_{t,a} = \frac{1}{H_x \times H_y} \times \sum_{i=1}^{i=R} \frac{h_x \times h_y \times T_{C,i}}{D_i}$$

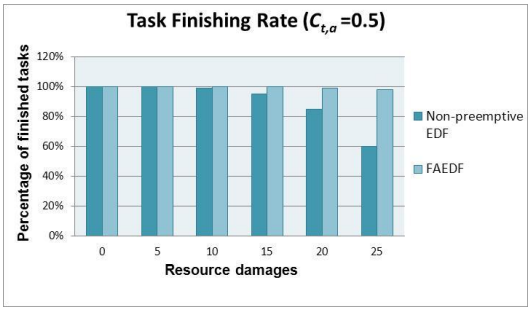
Equation 4-4 Average time-area constraint $C_{t,a}$

As opposed to the average time constraint C_t, the average time-area constraint $C_{t,a}$ considers not only the time constraint but also the task's area consumption. For instance tasks of larger size will increase $C_{t,a}$ since they may not fit into currently available resources. In the extreme case, $C_{t,a}$ will approach one, if: 1) every task's computation duration $T_{C,i}$ is equal to its relative finishing deadline D_i, where tasks have to be placed as soon as they are released; and 2) all tasks have the same size as the chip, so that only one task can be placed at any one time. On the other hand, $C_{t,a}$ approaches zero if all task sizes are extremely small and their deadlines are much larger than their computation durations. By using $C_{t,a}$, strictness can be better calibrated to give a more equitable test bench to evaluate the quality of the algorithms [Hong2011b].

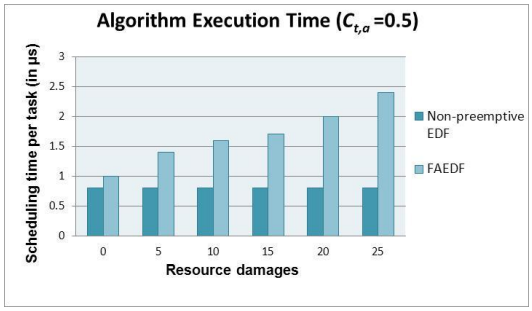
The following sections give the testing result of the C-language implementation of the algorithms on a desktop platform (Intel Core Due CPU at 2.4GHz). Three sets of

tasks are used to test the algorithms in three different *average time-area constraints* at 0.5, 0.75 and 0.9 respectively. Each task set contains 60 tasks and all of them are released at time zero. In our proposed test, all of the three sets of tasks are to be placed on a chip with a size of 15×12 CLBs, so that H_x =15 and H_y =12, which is an emulation of the CLB alignments on a Virtex-4 XC4VLX160 FPGA. To simulate faults, up to 25 CLBs are marked as damaged resources, which increases the difficulty in allocating a task. Both non-preemptive EDF and FAEDF use the EAC algorithm for allocation, and thereby only the quality of scheduling algorithms is compared with no complicating effect from the allocating algorithms. Figure 4.7, Figure 4.6, and Figure 4.8 give the simulation results for task finishing rate and algorithm execution time with increasing resource damage of 0 to 25 and average time-area constraints of 0.5 to 0.9.

In general, FAEDF outperforms non-preemptive EDF for task finishing rate when dealing with more damaged resources and strict time-area constraints; whereas non-preemptive EDF requires less algorithm execution time. Figure 4.7 shows that under low and average time-area constraints ($C_{t,a}$ =0.5), both non-preemptive EDF and FAEDF allow all tasks to finish when the number of damage is less than 15. However, when using non-preemptive EDF, the finishing rate decreases to ~60% when damage increases to 25, whereas FAEDF retains a similar rate at ~98%. On the other hand, the execution time of FAEDF increases linearly with the number of damaged resources, from ~1μs to ~2.4μs for scheduling one task, whereas non-preemptive EDF retains a constant time duration at about 0.8 μs. Figure 4.6 shows a similar pattern to that in Figure 4.7, where both task finishing rate and algorithm execution time change more dramatically compared to Figure 4.7. This trend is more obviously reflected in Figure 4.7, where high average time-area constraint ($C_{t,a}$ =0.9) is applied. In Figure 4.8, the task finishing rate dramatically decreases with increasing damage in both non-preemptive EDF and FAEDF, from ~80% to ~2% and ~65% to ~1% respectively, which indicates that FAEDF gives better scheduling efficiency when dealing with stricter deadline requirements.

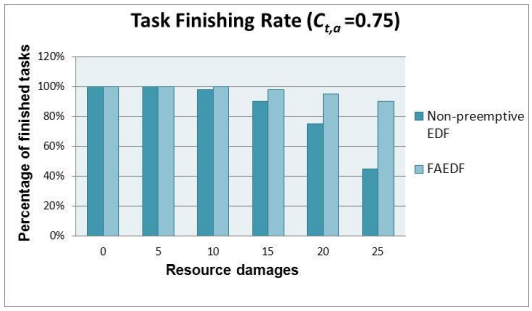


a) Task finishing rate in percentage

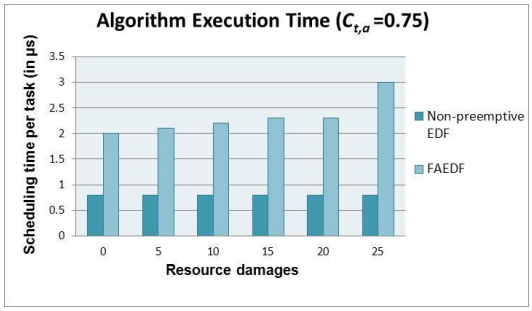


b) Algorithm execution time in µs

Figure 4.7 Simulation result of scheduling algorithms when $C_{t,a}$ is low

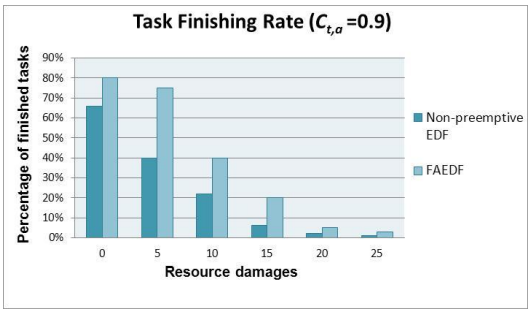


a) Task finishing rate in percentage

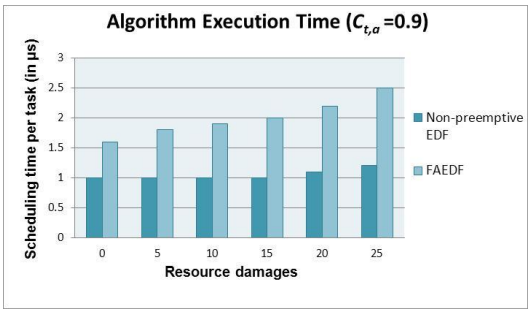


b) Algorithm execution time in µs

Figure 4.6 Simulation result of scheduling algorithms when $C_{t,a}$ is medium



a) Task finishing rate in percentage



b) Algorithm execution time in µs

Figure 4.8 Simulation result of scheduling algorithms when $C_{t,a}$ is high

4.2 Task Allocating Algorithms

The task allocation algorithms are used to improve the efficiency of usage of logic resources, by optimising the geometric placement of tasks to reduce chip area

fragmentation. Aiming at this objective, two allocating algorithms are developed; namely, the Empty Area Compaction (EAC) algorithm and the Empty Volume Compaction (EAV) algorithm. The following sections firstly give a summary of classical models for 2D-packing algorithms, and then introduce the proposed novel 2D-packing model and two innovative allocation algorithms. Finally, simulation results are presented to evaluate the performance of the proposed algorithms and to compare them with previous approaches.

4.2.1 Classical Models for 2D-packing

2D-packing algorithms can be differentiated depending on the way they describe their empty areas with an Empty Area Descriptor (EAD). In Chapter 2, most of the state-of-art allocation algorithms have been reviewed, including MER-based algorithms [Bazargan2000, Ahmadinia2004b, and Morandi2008], Vertex List Set (VLS) based analysis [Tabero2004, Ahmadinia2007], and staircase and line array approaches [Handa2004, Cui2007]. Among these, the MER-based and VLS-based algorithms are most commonly used, due to their easy explanation and satisfactory simulation results [Hong2011b]. This chapter further explains the MER and vertex approaches in terms of their area modelling and EADs, in order to compare them with the novel allocation algorithms proposed in this study.

The MER-based EAD uses Maximum Empty Rectangles (MERs) to keep track of all free areas in a two-dimensional space. The MERs can accommodate a upcoming task if its size is smaller than the size of MER, i.e. $h_{task,x} < h_{MER,x}$ and $h_{task,y} < h_{MER,y}$. The algorithm can choose either to use the first scanned possible MER or the smallest possible MER to accommodate the task after all MERs are scanned, which are termed First-Fit (FF) and Best-Fit (BF) respectively. After an MER is used to accommodate a task, it will be further partitioned into two smaller MERs, which are used to place later tasks. For example, in Figure 4.9.a, the whole chip area has only one MER (P) at the beginning. After task θ_1 is placed at MER P, the latter is fragmented into two smaller MERs P_1 and P_2. Afterwards, P_2 is used to place the next task θ_2 since it is smaller P_2. As a result, P_2 is further fragmented into P_3 and P_4. The advantage of this modelling is that the algorithm is easy to code, and the

algorithm execution time is a relatively constant value which does not increase dramatically with the accumulation of inserted tasks. However, as long as MERs are increasingly fragmented, they become too small to accommodate later tasks. In effect, the fragmented MERs do not reflect the real free area space on the chip, since they do not consider overlapping regions. For instance, in Figure 4.9.a, the region of P_4 can be partially shared by P_3 to make an MER larger than both P_3 and P_4.

In the light of this, the non-overlapped MER-based EAD is then extended to overlapped MER-based EAD (see Figure 4.9.b). In the overlapped MER-based EAD, the region of intersection can be merged and shared by multiple MERs, to create larger spaces in which to accommodate tasks. When a task is placed, the newly generated MERs will be compared with all previous MERs, and if there is any region which can be shared between different MERs, it will be merged into multiple MERs. For example, in Figure 4.9.b, after θ_1 is placed, the remaining two MERs (P_1 and P_2) have an overlapping intersection, and more intersections become overlapped after the placing of θ_2. By considering the overlapped regions, every MER can reflect the real maximum area size, whereby more tasks can be accommodated. However, since one region is contributing to multiple MERs, the updating of MERs becomes more complex and time consuming. If one MER is partitioned, all MERs that have an intersection with it will be affected and they have to be updated accordingly. Therefore, the algorithm execution time increases dramatically when more tasks are placed, causing a large time overhead. Large overheads in algorithm execution can delay task placement decisions; therefore, more tasks will miss their deadlines.

The VLS-based EAD records the free area information by using a list of vertices to describe the contours of free areas. All the vertices are used to place tasks, so that tasks can be placed at the corner of the areas, and hence less fragmentation is induced. The quality of placement is measured by the amount of adjacencies, where larger adjacencies give better compactness. For example, in Figure 4.9.c, there are originally four vertices (V_1 to V_4) before any task is placed. The first task θ_1 is placed at V_1, which gains three more vertices (V_5 to V_7). Afterwards, θ_2 is placed at V_7, where it can stay as close as possible to the area's contours. Note that during the

placement, the contour of the free area is kept as low as possible, to give more compactness for remaining free areas. The VLS-based EDA requires less time for computing positions and updating EDAs, and it is relatively easy to program. The idea is then further extended to three-dimensional VLS, in which not only are the 2D contours measured, but also time is taken into account in measuring the adjacency of areas rather than lines [Tabero2006]. However, one disadvantage of VLS-based EDA is that it only tracks the edges of free areas. As a consequence, it is not very efficient in coping with faults incurred in the middle of free areas [Hong2011a].

In the light of the aforementioned strengths and weaknesses of previous EADs, the new allocating algorithms are developed with novel EADs, whereby not only can the placement optimisation be achieved in a short time, but also emerging faults can be efficaciously coped with. The subsequent sections introduce two novel allocation algorithms, namely Empty Area Compaction (EAC) and Empty Volume Compaction (EAV) algorithm, which optimise task placement by maximising the remaining size of the empty rectangles.

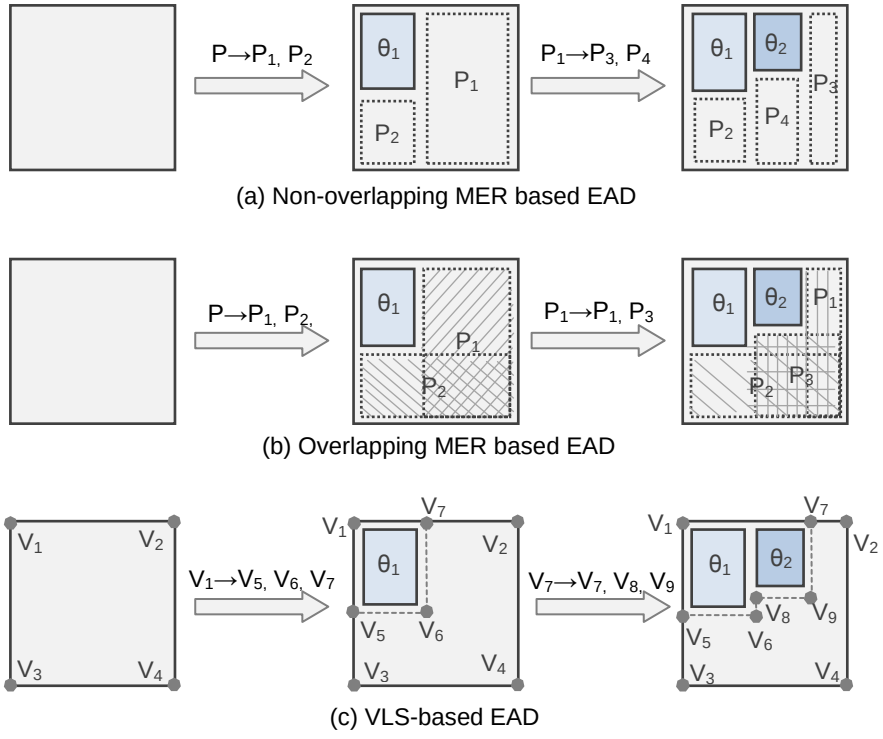


Figure 4.9 Three classical EADs

4.2.2 Empty Area Compaction (EAC) Algorithm

Like other 2D-packing algorithms, EAC uses an Empty Area Descriptor (EAD) to represent the array of CLBs in the FPGA, whereby occupation and damage status can be consecutively tracked and updated. The EAD evaluates the chip area as a rectilinear grid formed by cell arrays (CLB arrays). It uses two types of matrices to represent cell values; namely, a Shape-Matrix (SM) and an Area-Matrix (AM).

In the SM, both damaged and occupied CLB cells are labelled with zeros. The other CLB cells are labelled with sequential values starting from zero and incrementing horizontally in either the right or left direction (see Figure 4.10). An SM with a value incrementing from the left to right direction is called an SM-Left (SM-L); likewise an SM incrementing from right to left is named an SM-Right (SM-R). The SM-L is used to compute the width of the task to be allocated, which is one condition for placing this task at a position. For example, in Figure 4.10, the red shaded cell at position *(3, 4)* is a damaged CLB and the blue shaded cells are CLBs occupied by a task, which are all labelled with zeros. A task's position is defined as its bottom-right cell. For example, the blue shaded task has been placed at position *(3, 3)*, with a size of *2×2*. The cell value of SM-L indicates that only tasks with widths smaller than its value can possibly be placed at this position. For instance, the value of SM-L (b) at position *(2, 6)* is *3*, implying that if a task is to be placed at this position, its width has to be smaller than *3*. Note that, in order to place a task, only SM-L needs to be compared with the task width. Both SM-L and SM-R are used for computing AMs in the next stages.

Algorithm 4.3 and Algorithm 4.4 give the pseudo codes for computing SM-L and SM-R respectively. Both of these two algorithms use the current FPGA-state matrix as their input, in which all the occupied and damaged cells are marked as zeros. In SM-L, cells are marked with sequentially increasing numbers from left to right (lines 3 to 8). If the current cell is a damaged resource, or is occupied by a task, the value in SM-L will be reset to zero (line10). The function finally returns to the updated SM-L once all cells are scanned. The derivation of SM-R is similar to that of SM-L, with the only difference being that the value increases from right to left.

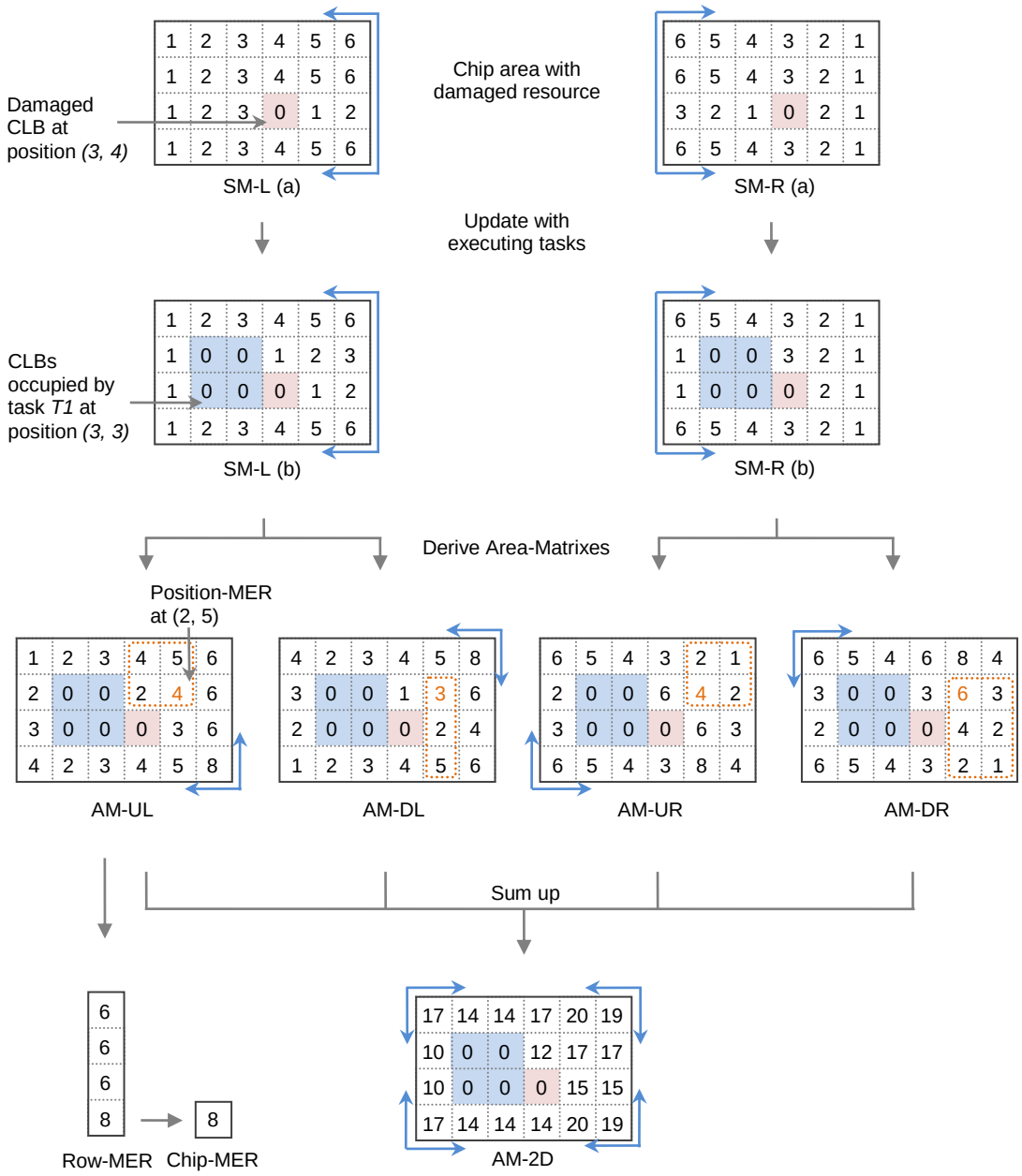


Figure 4.10 Derivation of shape and area matrices

AMs are derived from SMs. The cell value in AMs is called the position-MER, which represents the MER area in one of the four directions. There are four directional AMs and one sum AM (AM-2D). The four directional AMs are AM-Up-Left (AM-UL), AM-Down-Left (AM-DL), AM-Up-Right (AM-UR) and AM-Down-Right (AM-DR), where their position-MERs represent the MER areas in their up-left, down-left, up-right, and down-right directions respectively. For example, in Figure 4.10, at position *(2, 5)*, AM-UL gives a cell value of *4*, which indicates that the MER

Inputs:

Matrix *FPGA_state* showing placed tasks and damaged resources

Outputs:

SM-L

```
1.  begin:
2.    for( i = 1, i ≤ Hx, i++ )
3.      for( j = 1, j ≤ Hy, j++ )
4.        if (FPGA_state [i][j] ≠ 0)
5.          if ( j = 1)
6.            SM-L [i][j] = 1;
7.          else
8.            SM-L [i][j] = SM-L [i][j-1] + 1;
9.        else
10.         SM-L [i][j] = 0;
11.   return SM-L;
12. end
```

Algorithm 4.3 Derivation of SM-L matrix

Inputs:

Matrix *FPGA_state* showing placed tasks and damaged resources

Outputs:

SM-R

```
1.  begin:
2.    for( i = 1, i ≤ Hx, i++ )
3.      for( j = Hy, j ≥ 1, j-- )
4.        if (FPGA_state [i][j] ≠ 0)
5.          if ( j = Hy)
6.            SM-R [i][j] = 1;
7.          else
8.            SM-R [i][j] = SM-R [i][j+1] + 1;
9.        else
10.         SM-R [i][j] = 0;
11.   return SM-R;
12. end
```

Algorithm 4.4 Derivation of SM-R matrix

area (dashed encircled area) in its up-left direction is *4*. Likewise, the other three directional AMs, AM-DL, AM-UR, and AM-DR, give cell values of *3, 4,* and *6* to represent the MER area (dashed encircled areas) in its down-left, up-right, and down-right directions respectively. Central to this, the cell value in AM-UL implies the maximum task size (width × height) that can be accommodated at this position. For examples, position *(2, 5)* is an invalid position for tasks with areas larger than *4*. A task can possibly be placed at a position if three conditions are met: 1) the task area (width × height) is smaller than the value in AM-UL; 2) the task width is less than the value in SM-L, and 3) the values of all cells in AM-UL are not zero.

The derivation of directional AMs is based on a "windows expanding" method to compute each positional MER. Figure 4.11 gives an example of computing one positional MER in AM-UL from a given SM-L matrix. Here, a 3×3 SM-L matrix is given, in which one damaged resource is marked as zero at position (2, 1), and the other cells are labelled with a number increasing from left to right. In order to compute the positional MER, all the continuous non-zero positions in the same column, but above that position are assigned with a window. For example, to compute the MER of position (3, 3), which is the maximum empty rectangle in its top-left direction, positions (3, 3), (2, 3) and (1, 3), are assigned with a window each, which are used to measure the maximum area of each line. The window has a width which can be obtained directly from SM-L, and a height which starts from 1 and expands in both up and down directions. For example, in Figure 4.11.b, position (2, 3) has a window with a width of 2, which is its value in SM-L. The height of the window expands in both up and down directions, and increments its value from 1 to 3. However, the window stops expanding when a smaller value in the same column of SM-L is reached. For example, in Figure 4.11.a, the window cannot expand since the value at position (2, 3) is 2, which is smaller than the value "3" at position (3, 3). On the other hand, the width of the window cannot be larger than the width of the windows below. For example in Figure 4.11.c, the window's width is 2 rather than 3, since the width of the window below is 2. Finally, after the sizes of all windows are obtained, the position-MER is assigned with the maximum window's size, so that the position MER at (3, 3) is 6.

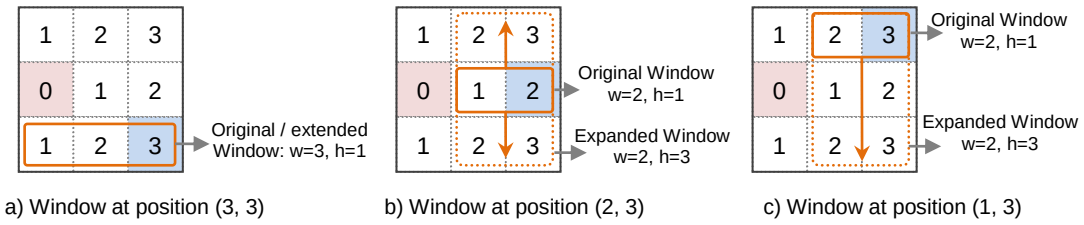


Figure 4.11 Computing the MER of position (3, 3)

Algorithm 4.5 gives the pseudo code to compute AM-UL and AM-DL matrices from SM-L. The algorithm is coded in line with the "window expanding" heuristic. Here, all the position-MERs are computed sequentially (lines 2 and 3) by measuring their expanded windows. For each position, all the cells above are scanned and their window sizes are computed (lines 7 to 25). The scanning will stop if a value of zero is reached (lines 9 and 10). To compute the size of the window, both the height and the width are calculated. The height of the window is reset to 1 at the beginning (line 8) and then expanded in both up and down directions (lines 13 to 17, and 18 to 22) if the value in SM-L is smaller than the value of the window's position (lines 14, 15, 19 and 20); otherwise it stops expanding (lines 16, 17, 21 and 22). If the width of the window is equal to less than the size of its previous windows' widths, it will be updated with the current window's width (lines 11 and 12); otherwise, it remains with the minimum width of the windows below. The final size of the window is obtained by multiplying its width and height (line 23), and the maximum window size will be chosen to be the final positional MER (lines 24 and 25). The function eventually returns to the updated AM-UL matrix. The AM-DL matrix is also derived from SM-L, with the difference that the windows are assigned to the cells below rather than the cells above the current position. The AM-DL matrix can be obtained by replacing lines 7, 13 and 18 with its comments (after the semicolon) in Algorithm 4.5. Likewise, the other two directional matrixes, AM-UR and AM-DR, can also be derived in the same way but using SM-R as their input, thus replacing SM-L, AM-UL, and AM-DL in Algorithm 4.5 with SM-R, AM-UR, and AM-DR respectively.

In addition, in order to accelerate the computation for selecting possible positions, Row-MER is used to preselect the scanning rows. The values in Row-MER are the

Inputs:

SM-L

Outputs:

AM-UL, AM-DL

```
1.  begin:
2.    for( i = 1, i ≤ Hx, i++ )
3.      for( j = 1, j ≤ Hy, j++ ) {
4.        AM-UL[i][j] = 0;
5.        window_size [i][j] = 0;
6.        width  = SM-L[i][j];
7.        for( m = i, m< Hy, m++ )  {          // AM-DL: for( m = i, m< Hy, m++ )
8.          height = 1;
9.          if (SM-L[m][j] ==0)
10.           break;
11.         if (width > SM-L[m][j] )
12.           width = SM-L[m][j];
13.         for( n = m, n< Hy, n++ )          // AM-DL: for( n = m, n> 0, n-- )
14.           if (SM-L[n][j] > SM-L[m][j] )
15.             height + +;
16.           else
17.             break;
18.         for( n = m, n> i, n-- )           // AM-DL: for( n = m, n< i, n++ )
19.           if (SM-L[n][j] > SM-L[m][j] )
20.             height + +;
21.           else
22.             break;
23.         window_size [i][j]= width × height ;
24.         if (window_size [i][j] > AM-UL[i][j])
25.           AM-UL[i][j]= window_size [i][j];
26.       }
27.     }
28.   return AM-UL;
29. end
```

Algorithm 4.5 Derivation of AM-UL and AM-DL matrixes

greatest position-MERs in each row, which are used to compare with the task width before scanning cells in that row. For instance, if a task width is larger than the current Row-MER, it will directly jump to the next row and all cells in that row will be omitted. Moreover, the greatest value in the Row-MER is called the Chip-MER,

which is the maximum affordable task size which can be placed on the current chip. The Chip-MER is fed back to the task scheduler for pre-selection; hence, the scheduling overhead is reduced since only feasible tasks are scheduled.

In order to evaluate the quality of a placement, AM-2D is used, which is derived by summing all of the four directional AMs, so that the value of AM-2D at position *(2, 5)* is *17*, which is the sum of AM-UL (*4*), AM-DL (*3*), AM-UR (*4*) and AM-DR (*6*). A larger value of AM-2D implies that this cell is contributing to more large MERs, and therefore its use should be avoided in order to retain larger MERs for later coming tasks. The intention is to place a task at a position where the sum of occupied cell values is at a minimum. The placement cost of a position is defined in Equation 4-5, where h_x and h_y are the task width and height respectively. A lower placement cost can achieve a better quality of allocation, so that more compact free spaces are left for later coming tasks.

$$Cost(x, y) = \sum_{i=x-h_x, j=y-h_y}^{i=x, j=y} AM - 2D[i, j] \qquad \textbf{Equation 4-5 Placement cost of (x, y) AM-2D}$$

The placement cost is calculated only if this position can possibly to accommodate the task. Afterwards, the position with the minimum placement cost is certified as the Best-Fit (BF) position. The BF position will give the optimum position at which to allocate a task, whereby more compact areas are retained and fragmentation is reduced. Alternatively, in order to further accelerate selection, First-Fit (FF) placement can be used, in which the first possible position is chosen to allocate the task and all other positions are bypassed.

Figure 4.12 gives the steps used to find the best position when allocating a task with the EAC allocation algorithm. Firstly, only the tasks with sizes smaller than that of the Chip-MER are scheduled by the task scheduler. Afterwards, the task allocator receives the prioritised task and compares it with Row-MER. If the task's area is larger than that of Row-MER, the current row will be bypassed; otherwise, all the position-MERs in the current row will be compared with the task's area. The task

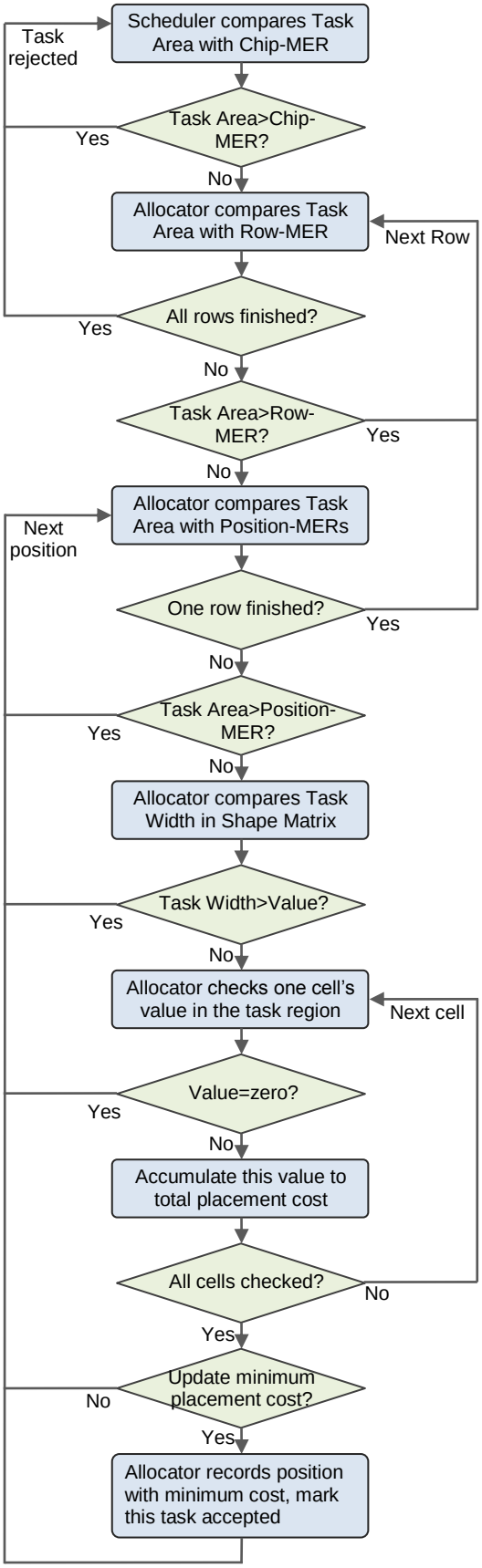


Figure 4.12 Steps to find a best-fit position to place a task

with an area smaller than the position-MER will be further compared with the value in SM-L to check if its width can fit into this position. Once the width fits, all the position-MERs occupied by the task will be checked, and if none of them is zero, they will be summed up to obtain the placement cost. After all the costs of each possible position are obtained, the position with the minimal cost will be selected to allocate the task to.

The pseudo code of the EAC allocation algorithm is presented in Algorithm 4.6. Corresponding to the flow chart in Figure 4.12, the algorithm outputs the best placement position, by using the given inputs, including Chip-MER, Row-MER, SM-L, AM-UL, AM-2D and task size. The algorithm scans all the positions on the chip in both vertical and horizontal directions (lines 3 and 5) and then filters the possible locations by sequentially comparing the task size with Chip-MER, Row-MER, AM-UL, and SM-L respectively (lines 2, 4, 6, and 7). Afterwards, all the values of the cells occupied by the task in AM-2D are summed to give a placement cost (line 11). If a zero value is detected, the cost value will reset to zero to indicate that the current position cannot accommodate the task and therefore it should be bypassed (lines 13 and 14). If the cost is not zero once all of the occupied cells are checked, it will be certified that the current position could possibly accommodate the task (line 16). In such a case, the cost is compared with the minimum cost that was previously calculated (line 17). If the current cost is less than the minimum cost, the minimum cost as well as its position will be updated (lines 18 and 19), and the cost will be reset to zero for the next computation (line 20). After all positions are scanned, the function will return to the best-fit position if one exists; that is, where the minimum cost has been updated at least once (lines 24 and 25), otherwise it returns to *null* (lines 26 and 27).

Inputs:

1. *Chip_MER*
2. *Row_MER [1,2,3…H_x]*
3. *SM_L [H_x, H_y]*
4. *AM_UL [H_x, H_y]*
5. *AM_2D [H_x, H_y]*
6. Task size $h_x \times h_y$

Outputs:

Best Fit position *[P_x, P_y]*

```
1.  begin:
2.    if (hx× hy < Chip_MER) {
3.      for ( i = 1, i< Hx, i++ )
4.        if  (hx× hy ≤Row_MER[i])
5.          for ( j = 1, j< Hy, j++ )
6.            if (hx× hy ≤AM_UL [i, j])
7.              if (hx ≤SM_L [i, j]) {
8.                for ( m= i, m< i - hy, m-- )
9.                  for ( n= j, n< j - hx, n--)
10.                   if (AM_UL [m, n] ≠ 0)
11.                     Cost = Cost + AM_2D[m, n];
12.                   else {
13.                     Cost = 0;
14.                     break;
15.                   }
16.               if (Cost ≠ 0)
17.                 if ( Cost ≤MinCost ) {
18.                   MinCost  = Cost ;
19.                   [Px, Py ] = [i, j];
20.                   Cost = 0;
21.                 }
22.             }
23.   }
24.   if ( MinCost ≠ 0 )
25.       return [Px, Py ];
26.   else
27.       return null;
28. end
```

Algorithm 4.6 EAC allocation algorithm

4.2.3 Empty Volume Compaction (EAV) Algorithm

The EAV is an extension of the EAC algorithm, with the additional consideration of task execution time, whereby the measurement of compactness can be extended from 2D area (area only) to that in the 3D volume (area and time).

Figure 4.13 gives an example where the efficiency of resource usage can be improved by placing a task in consideration of both time and area. In this example, both task1 (T1) and task2 (T2) are tasks already placed and in their execution state, and task3 (T3) is yet to be placed. When analysing placement using EAC in 2D, there is no difference between positions 1 and 2 (see Figure 4.13.a). However, a difference is shown when it is analysed in 3D. Since T2 finishes earlier than T1, at the time T2 is finished and removed, the remaining area will be more compact if T3 is placed at position 1.

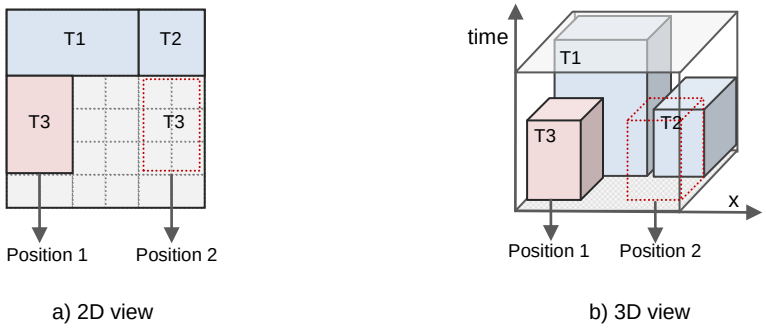


Figure 4.13 Task placement in consideration of time

Inspired by above, the EAV algorithm is designed to assign different weights to the edges of damaged resources as well as occupied cells (i.e. CLBs). The weight is determined by the remaining time duration of the damage or occupation from adjacent cells. A higher weight indicates that its adjacent cells will be occupied by more tasks for a longer time, whereas a lower weight implies that adjacent cells are occupied by fewer tasks and they are likely to be freed sooner. All the weights together form a Time Matrix (TM), in which all the cells are labelled with the sum of the remaining execution time of adjacent tasks, and the weight of permanent damage and chip edges. The weights of permanent damage and chip edges are determined by

the maximum remaining execution time of currently executing tasks. Figure 4.14 gives an example of the generation of the TM from the original FPGA state matrix. The FPGA state matrix describes the current state of the chip, which is allocated to task1 (T1) and task2 (T2). T1 occupies cells (2, 2) and (3, 2), with a remaining execution time of 3, whereas T2 is allocated to cells (2, 3) and (3, 3), with a remaining execution time of 2. In addition, a damaged CLB is detected at position

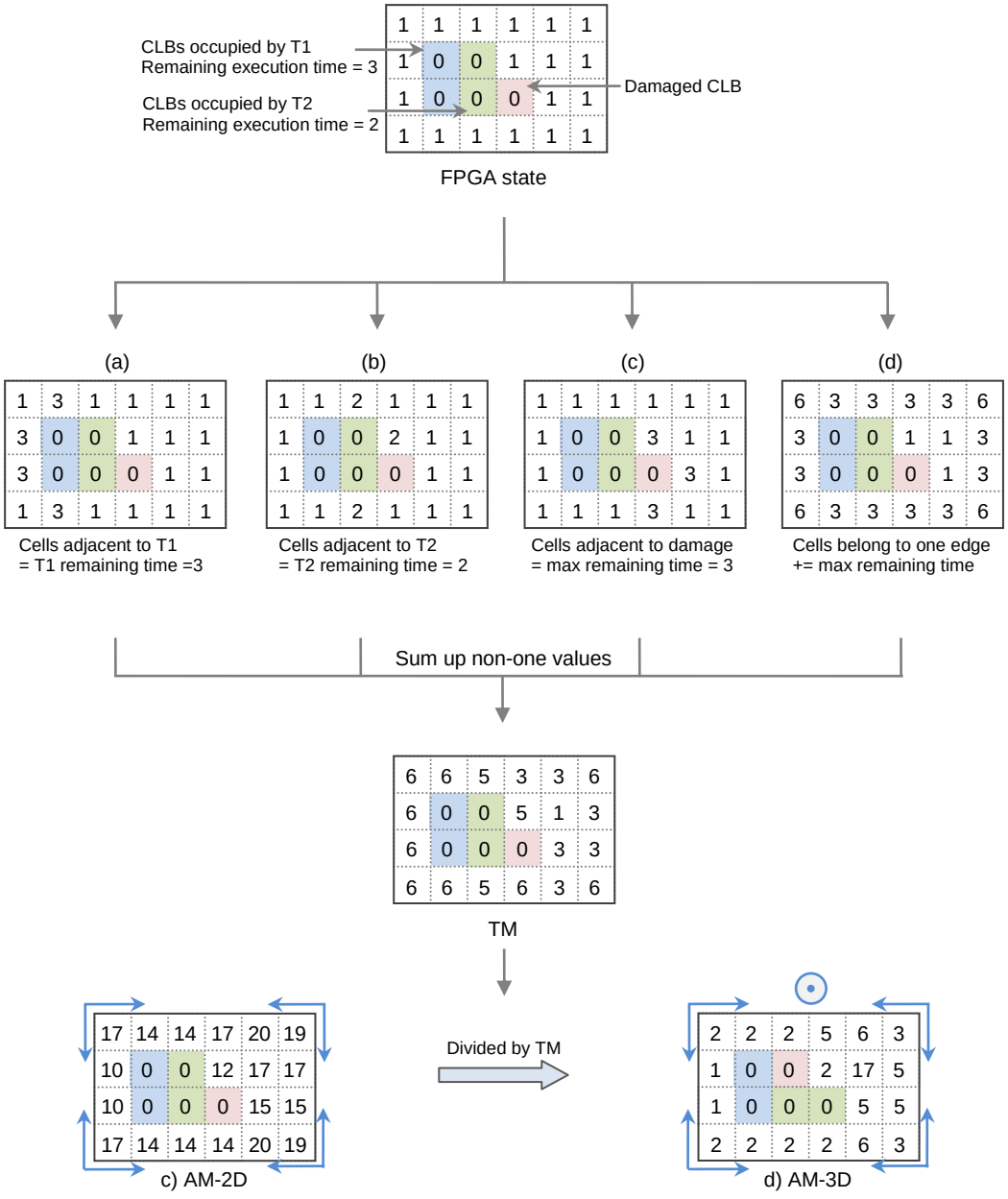


Figure 4.14 Derivation of AM-3D matrix

(3, 4). In order to generate the TM from the FPGA state matrix, all the cells adjacent to a task are firstly marked with the remaining execution time of that task (see Figure 4.14, matrices (a) and (b)). In addition, all the cells adjacent to a damaged resource are marked with the maximum remaining execution time of all the tasks, so that T1 has the maximum remaining execution time of 3, and so the cells adjacent to the damaged CLB are marked with 3 (see Figure 4.14, matrix (c)). Moreover, all the cells at the chip edges are marked with the maximum remaining execution time, and the cells belong to two edges, i.e. a chip corner, so that this is twice the maximum remaining execution time (see Figure 4.14, matrix (d)). Finally, all the incremented values (non-one values) are accumulated to generate the TM. The TM gives information about task adjacencies in the time domain, which is further used to generate the three-dimensional area matrix, namely the AM-3D. To derive the AM-3D, all of the values in the AM-2D are divided by the values in the TM. The resultant AM-3D matrix (with values rounded) is then used to compute the placement cost. The EVC algorithm differs from the EAC algorithm only in its cost function, which uses the value in the AM-3D rather than the AM-2D.

Algorithm 4.7 gives the pseudo code to derive the TM matrix. The algorithm inputs include the FPGA state, which describes the current resource availability of the chip in terms of occupied tasks and damaged resources. The value of each cell (the weight) is computed by sequentially checking its adjacent cells in four directions (lines 4 to 31). In each direction, it is firstly check if the cell is at the edge of the chip (lines 4, 11, 18, and 25). If it is, the value will be accumulated with the maximum remaining executing time (lines 10, 17, 24, and 31); otherwise, adjacent tasks and damaged resources are checked. If the cell is adjacent to a damaged resource or an executing task, its value will be accumulated with either the maximum remaining executing time (lines 6, 13, 20, and 27), or the remaining executing time of the occupied task (lines 8, 15, 22, and 29). After all the cells have been checked, the function returns to the updated TM matrix.

Algorithm 4.8 presents the function used to generate the final AM-3D matrix from the TM and AM-2D matrices. If the value in TM is not zero, the value in AM-3D is

Inputs:

1. Matrix *FPGA_state* showing placed tasks and damaged resources
2. Executing task queue E
3. The maximum remaining executing time T_m

Outputs:

Time matrix *TM*

```
1.  begin:
2.    for( i = 1, i ≤ Hx, i++ )
3.      for( j = Hy, j ≥ 1, j-- )
4.        if ( i > 1 )
5.          if (FPGA_state [i-1][j] is Damaged )
6.            TM [i][j] = TM [i][j] + Tm ;
7.          else if (FPGA_state [i-1][j] is occupied by task Tk )
8.            TM [i][j] = TM [i][j] + remaining executing time of Tk ;
9.        else
10.           TM [i][j] = TM [i][j] + Tm ;
11.       if ( i < Hx)
12.         if (FPGA_state [i+1][j] is Damaged )
13.           TM [i][j] = TM [i][j] + Tm ;
14.         else if (FPGA_state [i+1][j] is occupied by task Tk )
15.           TM [i][j] = TM [i][j] + remaining executing time of Tk ;
16.       else
17.           TM [i][j] = TM [i][j] + Tm ;
18.       if ( j > 1 )
19.         if (FPGA_state [i][j-1] is Damaged )
20.           TM [i][j] = TM [i][j] + Tm ;
21.         else if (FPGA_state [i][j-1] is occupied by task Tk )
22.           TM [i][j] = TM [i][j] + remaining executing time of Tk ;
23.       else
24.           TM [i][j] = TM [i][j] + Tm ;
25.       if ( j < Hy )
26.         if (FPGA_state [i][j+1] is Damaged )
27.           TM [i][j] = TM [i][j] + Tm ;
28.         else if (FPGA_state [i][j+1] is occupied by task Tk )
29.           TM [i][j] = TM [i][j] + remaining executing time of Tk ;
30.       else
31.           TM [i][j] = TM [i][j] + Tm ;
32.   return TM;
33. end
```

Algorithm 4.7 Derivation of TM matrix

the quotient of the value in AM-2D divided by the value in TM; otherwise, it is zero. The function returns to the AM-3D matrix once all cells are updated.

Inputs:
TM, AM-2D
Outputs:
AM-3D

```
1.  begin:
2.    for( i = 1, i ≤ Hx, i++ )
3.      for( j = 1, j ≤ Hy, j++ )
4.        if (TM [i][j] ≠ 0)
5.          AM-3D [i][j] = AM-2D[i][j] / TM [i][j]
6.        else
7.          AM-3D [i][j] = 0;
8.    return AM-3D;
9.  end
```

Algorithm 4.8 Derivation of AM-3D matrix

After the AM-3D matrix is computed, the following allocation steps are exactly the same as these with the EAC algorithm, except that the placement cost is calculated using the values in the AM-3D instead of the AM-2D, shown in Equation 4-6.

$$Cost(x, y) = \sum_{i=x-h_x, j=y-h_y}^{i=x, j=y} AM - 3D[i, j] \qquad \textbf{Equation 4-6 Cost (x, y) of AM-3D}$$

4.2.4 Simulation Results of Allocating Algorithms

To test the allocation algorithms, two tests are simulated to show the improvements achieved by the EAC algorithm in comparison with previous algorithms, and also the difference between the EAC and EAV algorithms. The first test gives an evaluation on the task finishing rate, which is increased by using the EAC algorithm compared with previous algorithms. The second test compares the EAC and EAV algorithm, in respect of their task finishing rates and algorithm execution times. All of the

algorithms are programed and executed on an Intel Core Duo CPU at 2.4GHz and their simulation results are presented below.

In the first test, the EAC algorithm is compared with the two classical allocation algorithms, namely the overlapping MER-based algorithm and the VLS-based algorithm. In order to better explore the efficacy of the allocation algorithms, they are tested in a finer granularity at 100×100 CLBs. There are six task sets to be tested, and each contains 500 tasks of various sizes. TABLE 4.2 lists the parameters of the six task sets (from φ_1 to φ_6), where the width and height of a task is a random number ranging from $h_{x,min}$ to $h_{x,max}$, and from $h_{y,min}$ to $h_{y,max}$ respectively, so that, for example, in task set φ_1, a task size varies from 1×1 to 10×10. In order to test the pure allocation efficacy under the same benchmark, all the tasks are placed at the same time, so that all tasks have the same arriving time at 0, and expire immediately if they cannot be placed the first time. Figure 1.1 shows the overall task finishing rate, which is the number of allocated tasks divided by the number of unplaced tasks, of the three algorithms when there is no damaged resource on the chip. On average, the EAC algorithm outperforms the other two algorithms, with ~25% improvement in the task acceptance rate. Figure 4.16 shows the task finishing rate of each task set, with an increasing number of damaged resources. In all task sets, the task finishing rates of the other two algorithms (MER and VLS) decrease dramatically with increases in damaged resources, whereas the task finishing rate of EAC decreases less than the other two. This implies that the EAC is more efficacious in dealing with faulty resource allocation.

TABLE 4.2 SIX TASKS SETS WITH DIFFERENT PARAMETERS

	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6
$h_{x,max}$	10	10	20	20	40	40
$h_{x,min}$	1	5	1	5	1	5
$h_{y,max}$	10	10	20	20	40	40
$h_{y,min}$	1	5	1	5	1	5
$H_x \times H_y$	100×100 CLBs					
n_{task}	500 tasks					

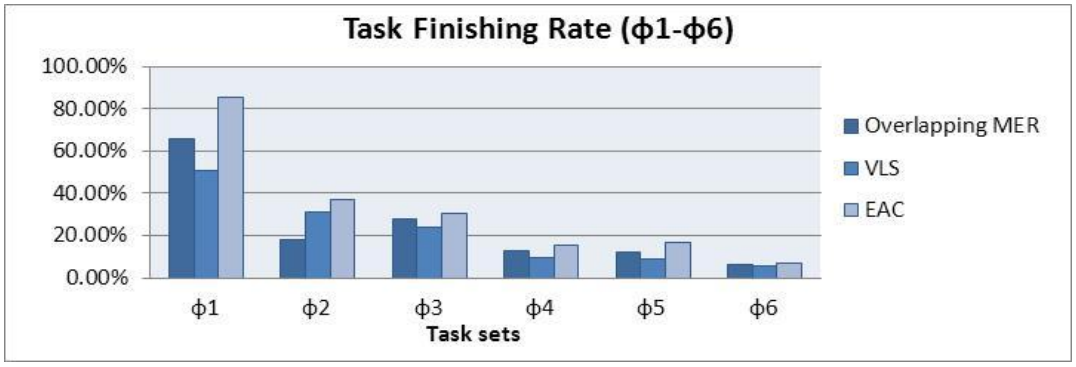
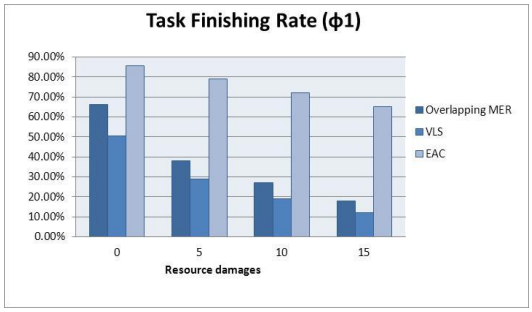
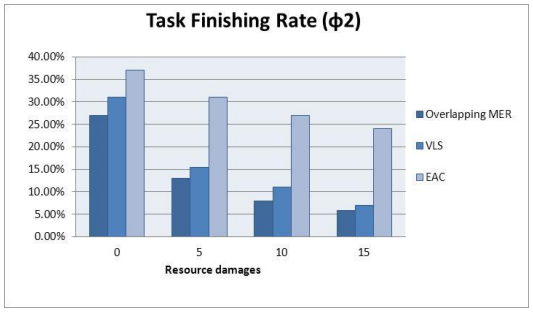


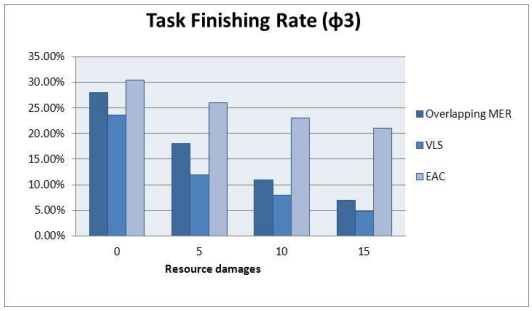
Figure 4.15 Task finishing rate of non-damaged resources



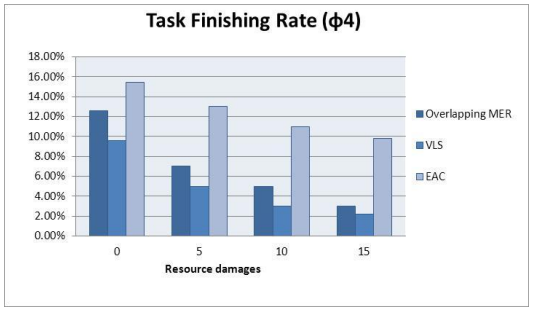
a) Task set φ_1 finishing rate in percentage



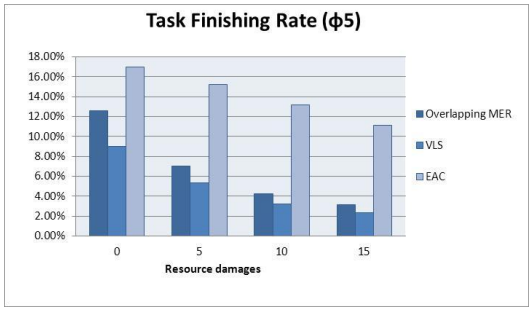
b) Task set φ_2 finishing rate in percentage



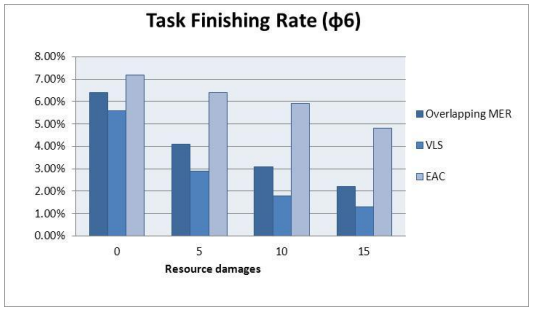
c) Task set φ_3 finishing rate in percentage



d) Task set φ_4 finishing rate in percentage



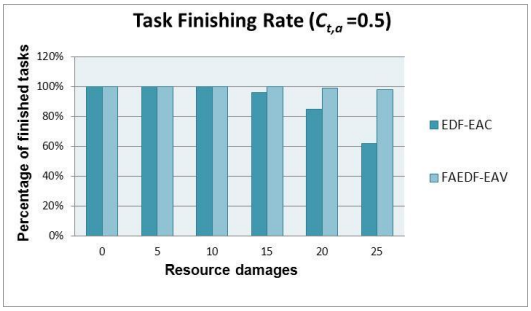
e) Task set φ_5 finishing rate in percentage



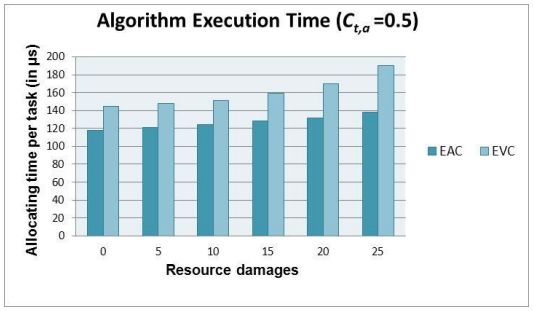
f) Task set φ_6 finishing rate in percentage

Figure 4.16 Task finishing rate of damaged resource

In the second test, the EAC algorithm is compared with the EAV algorithm, in respect of their task finishing rates and algorithm execution times. In order to show the difference between the 2D analysis and the 3D evaluation, the EDF scheduling algorithm and the EAC allocation algorithm are combined together to give 2D-based task managing; likewise, the FAEDF scheduling algorithm and the EAV algorithm are integrated to give full 3D volume-based task management. The execution time of the allocation algorithm includes the time to find the best location, as well as the time to update all the matrices. Figure 4.17, Figure 4.18, and Figure 4.19 give the task finishing time and algorithm execution time with three different values of *average time-area constraint* $C_{t,a}$. In terms of the task finishing rate, the 3D-based FAEDF-EVC achieves better results than the 2D-based EDF-EAC algorithm. This trend becomes more significant when more damaged resources are detected, or a stricter time-area constraint is applied. On the other hand, the execution times of both of the algorithms increase linearly when there are more damaged resources; however, the increasing rate of increase for the EVC is higher than that for the EAC; hence, the EVC's average execution time is higher than that of the EAC.

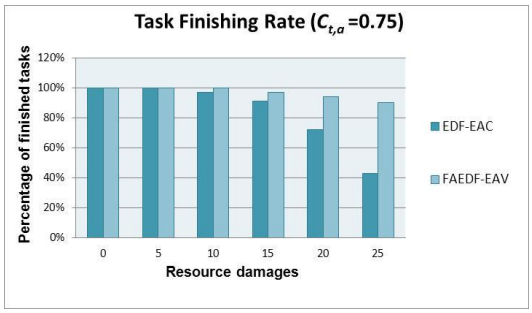


a) Task finishing rate in percentage

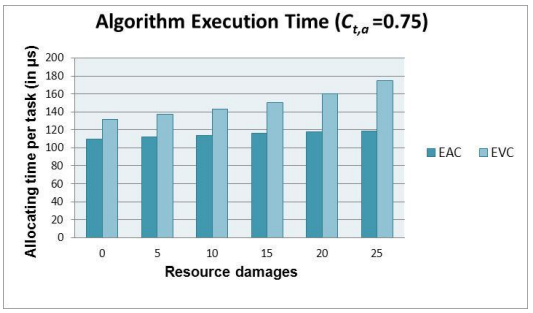


b) Algorithm execution time in µs

Figure 4.17 Simulation result of allocating algorithms when Ct,a is low

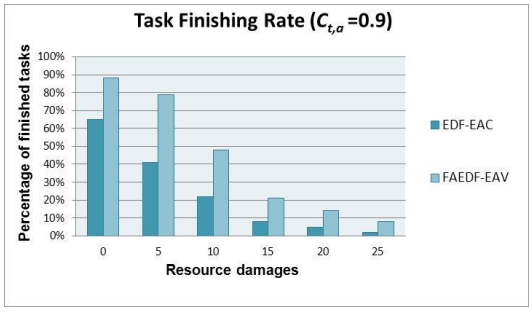


a) Task finishing rate in percentage

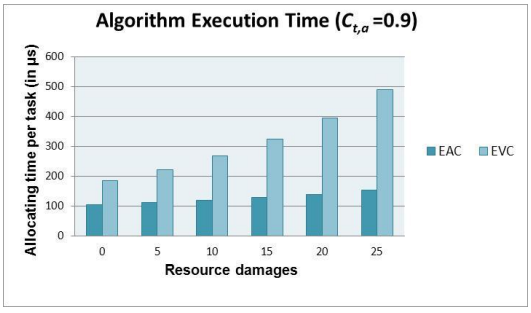


b) Algorithm execution time in µs

Figure 4.18 Simulation result of allocating algorithms when Ct,a is medium



a) Task finishing rate in percentage



b) Algorithm execution time in µs

Figure 4.19 Simulation result of allocating algorithms when Ct,a is high

4.3 Conclusion

This chapter has presented two scheduling algorithms, namely the EDF and FAEDF algorithms, and two allocation algorithms, the EAC and EAV algorithms. The

FAEDF is a novel scheduling algorithm which extends the traditional EDF by taking into consideration task finishing time, whereby tasks can be more efficiently scheduled. The EAC is an innovative allocation algorithm which optimises the placement of tasks by measuring the remaining MERs on the chip, while the EAV algorithm extends the EAC by adding time analysis to optimise placement in 3D volumes.

The simulation results show that both of the proposed scheduling and allocation algorithms achieve better task finishing rates compared with previous algorithms, with lower overheads in terms of execution time. Moreover, the proposed algorithms are more efficacious in dealing with damaged resources, with less updating time needed and higher placement rate achieved.

It is also acknowledged that both of the scheduling and allocating algorithms are not strictly restricted to the context of R3TOS. In effect, both of the scheduling algorithm (FAEDF) and allocating algorithm (EAC and EVC) can be widely applied for other contexts, where tasks can be modelled as blocks to be mapped to two-dimensional logic resources.

Trading off the simplicity of algorithm coding against its performance efficacy, the EDF & EAC algorithms are chosen to be implemented in the R3TOS. The detailed implementation of the scheduling and allocation algorithms, as well as other hardware modules, are presented in the next chapter.

Chapter 5

Hardware Implementation

This chapter presents the implementation of the low level hardware in the R3TOS, giving technical support to the algorithms and methodologies introduced in the previous chapters. The R3TOS implementation consists of four major components namely; the task scheduler, the task allocator, the ICAP manager and the host API. The task scheduler and allocator are responsible for scheduling tasks in real time and allocating tasks to optimum positions, whereby more tasks can meet their deadlines and hardware resources are more efficiently used. The previously presented EDF scheduling and EAC allocation algorithms are coded in the task scheduler and task allocator respectively. The ICAP manager provides the low level services that directly communicate with the ICAP port and the configuration memory, including writing/reading bitstreams to/from the ICAP port, as well as bitstream manipulation. As presented in Chapter 3, the above three components, namely the scheduler, allocator and ICAP manager, together comprise the hardware microkernel (HWμK) of the R3TOS which facilitates the complex exploitation of the low level DPR hardware. The host API is the main CPU that executes user applications and serves as the highest level of the system, providing a generic API between the user application and the R3TOS HWμK.

Related publications: [Hong2011b], [Hong2012a] and [Hong2012b].

All of the three components in the HWμK are separately implemented on three ECC-protected fault-tolerant soft processors; namely, ECC-FT-processors. The development of the ECC-FT-processor is based on the Xilinx PicoBlaze processor, which is then enhanced by ECC-protected memories using the proposed novel hardware adaptor and recovery mechanisms. Thereby, the ECC-FT-processor can self-heal from an emerging fault without interrupting its operations, which considerably improves system reliability.

The following sections firstly present the design and implementation of the fault tolerant microprocessor. Afterwards, built on this processor framework, the specifications of the task scheduler, allocator, and ICAP manager are detailed, including their communication inputs and outputs, software coding, and management of memory space. Last, but not least, the design and implementation of the hardware based SMP communication architecture is presented. The work on overall system integration and host API was conducted by the research group member Xabier Iturbe [Iturbe2013b, Iturbe2013c]; and is therefore not included in this thesis. Note that the whole R3TOS system is implemented in cooperation with other group members, namely Xabier Iturbe and Ali Ebrahim. Xabier Iturbe implemented the host API, part of the communication mechanism and performed system integration, while Ali Ebrahim implemented the ICAP manager.

5.1 Design and Implementation of Fault-Tolerant Soft Processors

Nowadays FPGAs are widely used in many avionics and safety-critical aerospace applications, due to their increasingly high performance and flexible reconfigurability, which allows for in-system reconfiguration after launch [Katz2003]. At the same time, the functionalities of FPGAs are further enhanced by embedding on-chip microprocessors. The microprocessors can be embedded either in the form of built-in hard cores, such as the PowerPC and ARM-CortexA9 on Xilinx FPGAs, or soft cores using the reconfigurable logic resources of FPGAs, such as the PicoBlaze and MicroBlaze from Xilinx, and Nios from Altera. To date, an increasing number of applications have been updated onto microprocessors due to their

programing flexibility and close compatibility with high level user applications [Iturbe2010a].

One major concern is that most successful commercial FPGAs are based on SRAM memory technology, which suffers from high susceptibility to radiation-induced faults such as SEUs. In the light of this, the configuration memory of FPGAs is hardened so as to be more robust than the traditional SRAM, especially in some safety-critical applications. Nonetheless, block memory (BRAM) is still under the potential threat, since they are more compact and dense cells [Xilinx2011a]. According to the Rosetta experiment, BRAM blocks are at least double the risk from high energy particles in space [Lesea2005]. In the 90 nm technology Spartan-3 FPGA series, the error rate of CLBs is about 111 FIT/Mb, whereas for BRAMs it is 222 FIT/Mb, which is twice that of CLBs. More worryingly, the BRAM is about four times more susceptible to faults than other components in Virtex-4 family FPGAs, with 222 FIT/Mb compared to 61 FIT/Mb for other components [Lesea2005].

In order to circumvent this potential threat, Xilinx has developed hardware solutions which utilise Error-Correcting Code (ECC) circuitry to protect data in all BRAM blocks. This gives the ability to correct one single bit error, as well as to detect the location of double errors [Xilinx2008]. However, this ECC mechanism is not compatible for use as program memory in most on-chip microprocessors. The main difficulty arises with synchronisation between the processor and the program memory. For example, processors embedded in FPGAs typically require one clock cycle latency when reading data from its program memory, which is applicable to the traditional BRAMs on FPGAs. However ECC-BRAMs need two clock cycles to read one data, which does not satisfy the timing requirement of microprocessors. To solve this problem, a hardware interface is presented called EPA (the ECC Processor Adaptor) to synchronize the timing between the processor and ECC-BRAM by “looking ahead” to the next instruction [Hong2012a]. The EPA and LookAhead strategy is demonstrated in the context of the Xilinx 8-bit PicoBlaze microprocessor, due to its flexible implementation architecture. The following sections firstly give a brief introduction to the PicoBlaze microprocessor and ECC-BRAM. Then the

architecture of the EPA and its connections are presented, before the LookAhead strategy is illustrated. Finally the self-recovery mechanism and its testing results are presented.

5.1.1 PicoBlaze microprocessor and ECC-BRAM

PicoBlaze microprocessor

The PicoBlaze is an 8-bit Xilinx soft-core microprocessor [Xilinx2011b], which is favoured by developers for some simple protocol-based applications due to its low resource consumption. The processor core, that is, the processor state machines, only requires 96 slices when implemented on Xilinx Spartan-3, Virtex-II, and Virtex-II Pro FPGA architectures. Therefore it is widely used in some simple protocol-based applications, such as I_2C, SPI, PWM, UART protocols, as well as co-processor accelerators such as multipliers, DSP, and interrupt controller for MicroBlaze. Besides this, PicoBlaze is an open-source softcore processor, which gives the flexibility to adapt its architecture according to the demands of various applications. Moreover, programs in PicoBlaze are coded in the assembler, which makes optimised low-level programming possible. The basic interfaces of the PicoBlaze microprocessor are standard IOs, including the address bus (*Port_ID*), the unidirectional data buses (*In_Port* and *Out_port*), the signals to indicate W/R operation (*Write_Strobe* and *Read_Strobe*), interrupt signals (*Interrupt* and *Interrupt_Ack*), and the reset (see Figure 5.1). The program memory of the PicoBlaze is implemented in one BRAM block, which can store up to 1024 18-bit instructions. Figure 5.1 shows the standard configuration between the PicoBlaze core and its program memory, in which the Program Counter (PC) is sent to BRAM as its 10-bit memory address, and the 18-bit instruction is fetched by reading from the BRAM.

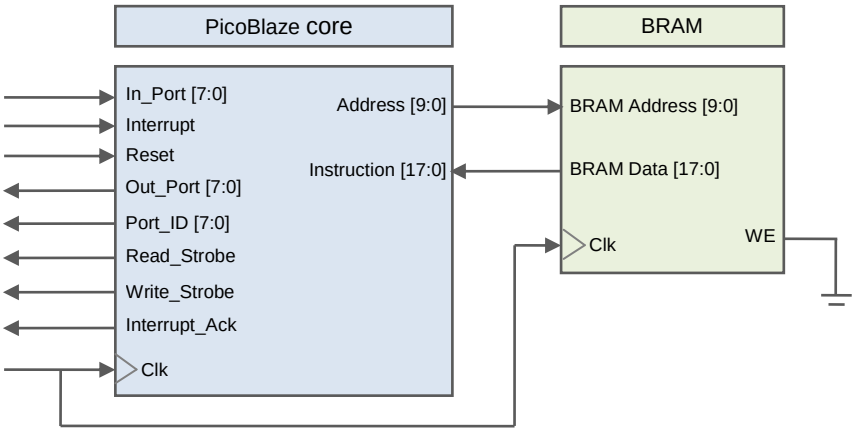


Figure 5.1 PicoBlaze standard configuration

For the standard PicoBlaze, each operation, including fetching, decoding and executing, takes a constant period of 2 clock cycles, and the instruction is executed one clock cycle after the program counter changes. Figure 5.2 gives the standard operation timing of the PicoBlaze. Here, the PicoBlaze address is the program counter that increments every two clock cycles when there is no branching. The program counter is connected to the BRAM's address bus, and the output data (PicoBlaze instructions) from the BRAM become available one clock cycle after the address is given, since the BRAM has a latency of one clock cycle. When a branching operation is executed, such as JUMP, the program counter changes one clock cycle after its execution, and the next instruction can be fetched consistently; therefore, there is no bubble created during program branching (see Figure 5.2).

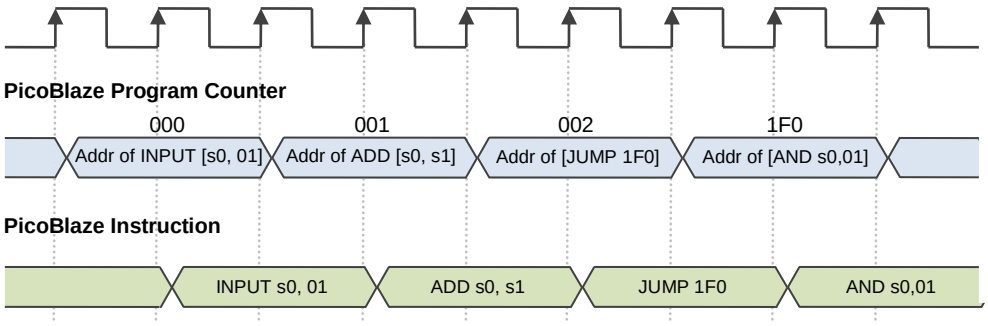


Figure 5.2 PicoBlaze standard operation timing

ECC-protected BRAM

The ECC-protected BRAM (ECC-BRAM) is composed of two standard BRAM blocks and an error correction encoder/decoder (see Figure 5.3). Each standard BRAM block has a 9-bit address bus and a 36-bit data bus. The 36-bit data is composed of 32-bit data and 4-bit ECC codes. Two 36-bit data buses are combined together to form a 72-bit data bus, which includes 64-bit data and 8-bit ECC codes. The 72-bit data bus is bidirectional, and is connected to both the ECC encoder and the ECC decoder. When writing data to the ECC-BRAM, the 8-bit ECC code will be automatically generated by the ECC encoder from the 64-bit user data, and then stored in the BRAM. During system operation, if a single bit in the ECC-BRAM is corrupted, the error bit will be automatically corrected by the decoder when it is read from the ECC-BRAM. At the same time, the error type can be indicated from a 2-bit error status register. The 4 possible errors types and their status are given in TABLE 5.1.

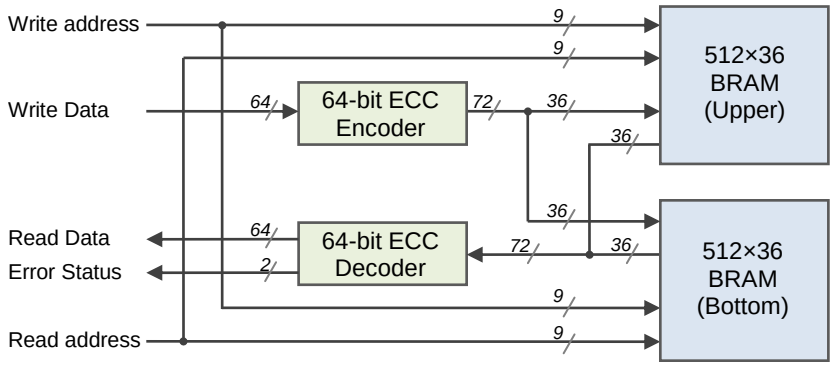


Figure 5.3 ECC BRAM internal structure

TABLE 5.1 ECC-BRAM ERROR STATUS

Error status	Type of error
00	no error
01	single error
10	double error
11	undefined invalid error status

However, code correction is provided at the cost of one clock cycle latency. The ECC-BRAM has a latency of two clock cycles, since one extra cycle is used by the decoder to decode the ECC code. Consequently, at least two clock cycles are needed to fetch one instruction from the ECC-BRAM, which is not compatible with PicoBlaze timing. Nevertheless, ECC-BRAM provides a higher data width (64-bit), which can contain up to three 18-bit instructions for the PicoBlaze. In the proposed implementation, two 18-bit instructions are packed together and stored in one 64-bit data in the ECC-BRAM, which is enough to hide its latency (see Figure 5.4). Note that the ECC-BRAM can also be applied to MicroBlaze. MicroBlaze uses 32-bit instructions, whereby two instructions can be stored in one address in the ECC memory.

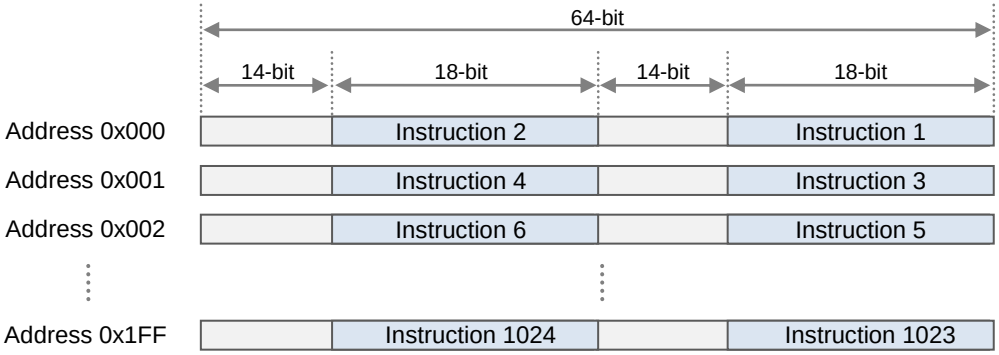


Figure 5.4 Instruction mapping in ECC-BRAM

5.1.2 ECC Processor Adaptor Configuration

The ECC Processor Adaptor (EPA) is the interface between the standard PicoBlaze and the ECC-BRAM, which solves the incompatibility in synchronisation by looking ahead to the next instruction in the ECC-BRAM. Figure 5.5 shows the connections of the EPA and the PicoBlaze processor core and ECC-BRAM. The basic IOs of the standard PicoBlaze remain unchanged, but both the 10-bit address output (PC) and the instruction input are connected to the EPA rather than the memory. Since the memory depth of ECC-BRAM is 9, whereas the PC has a data width of 10, the least significant bit of the PC is not sent to the ECC-BRAM, but is instead used to select one of the two fetched instructions. The two instructions (higher and lower) are

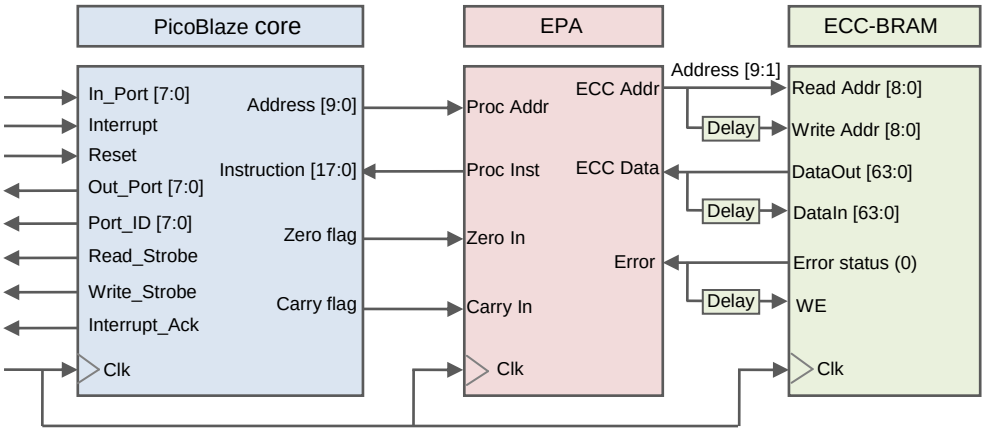


Figure 5.5 EPA connection

decoded by the lowest bit of the PC, where '1' is used to select the lower instruction and '0' for the higher instruction (see Figure 5.6). The data output of the ECC-BRAM is fed back to its input, and the lower bit of error status is connected to the memory's write enable signal, and thereby the faulty bit can be corrected automatically. The detailed timing of the recovery mechanism is illustrated in section 5.1.4 below. In addition, both the zero flag and carry flag signals are extracted from the processor core and used by the EPA to look ahead to the branching operations.

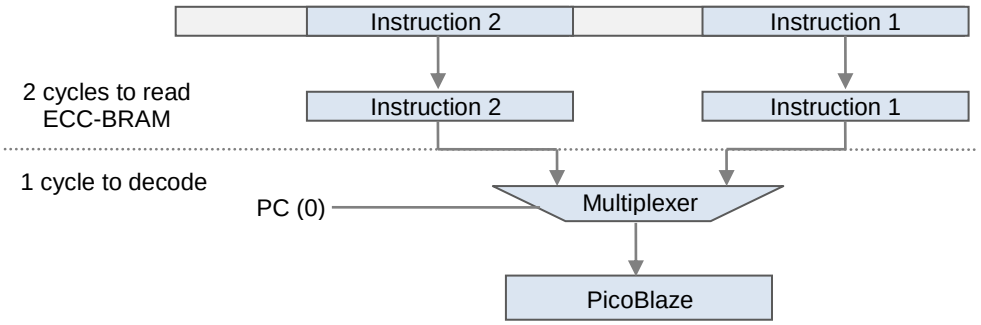


Figure 5.6 Instruction decode

5.1.3 Lookahead Strategy

When using the EPA and ECC-BRAM, a total of three clock cycles of latency are involved in fetching one instruction. This three clock latency includes two cycles for reading data from the ECC-BRAM (one for reading from the BRAM and one for the ECC decoding), and one cycle for decoding the EPA instruction; that is, multiplexing

two fetched instructions. As a result, the instruction is available three clock cycles after the PC changes, which is not compatible with processor timing. To solve this problem, the Lookahead strategy to pre-fetch and preempt desired instructions is presented, for use especially in situations where branching operations are executed. The detailed timing of the implementation is described in Appendix-1 (Operation Timing of Fault Tolerant Microprocessor), which illustrates the time for each branching operation, such as JUMP, CALL, RETURN, and INTERRUPT.

5.1.4 Self-Recovery mechanism

The proposed fault tolerant microprocessor has the ability to fix a single fault automatically without interrupting its operation, to detect multiple errors by using the TMR, and to recover from multiple errors by self-reconfiguring.

The processor can automatically recover from a single error, such as SEU, benefiting from its ECC-BRAM. As mentioned above, the ECC-BRAM can detect a double-error fault and correct a single fault when the data is read from the memory. In the proposed fault tolerant microprocessor, once a faulty bit is read from the memory it will be firstly corrected by the ECC decoder, and then written back to the same address in the memory, whereby the faulty data can be overwritten (see Figure 5.7). The first bit of the error status register, that is, the status (0), is connected to the write enable port of the ECC-BRAM (see Figure 5.5), since it indicates a single bit error (see TABLE 5.1). Besides this, the output data port of the ECC-BRAM is connected to its input data port, with a four-cycle delay, and therefore the corrected data can overwrite the faulty data once it is accessed from the ECC-BRAM. The four-cycle delay is used to avoid a situation where the same data is read and written at the same time, which represents an invalid access. In the extremely rare case of the PC remaining at the same address for more than four cycles, the memory will be illegally accessed, so that correction cannot be guaranteed, and further reconfiguration will be performed, such as system scrubbing.

Both the PicoBlaze core and the EPA are triplicated using TMR, and two voters are used to vote for the majority from the processor output (see Figure 5.8). The voter

will output an error signal if one of the three triplicated components gives a different output. On the other hand, the second bit of the error status signal, that is, the status (1), indicates that multiple errors exist in one data, which cannot be corrected. In the situation where either the status (1) or the voter is set, the whole processor will send a system error signal to indicate an unfixable error, in which case the whole microprocessor needs to be reconfigured by scrubbing.

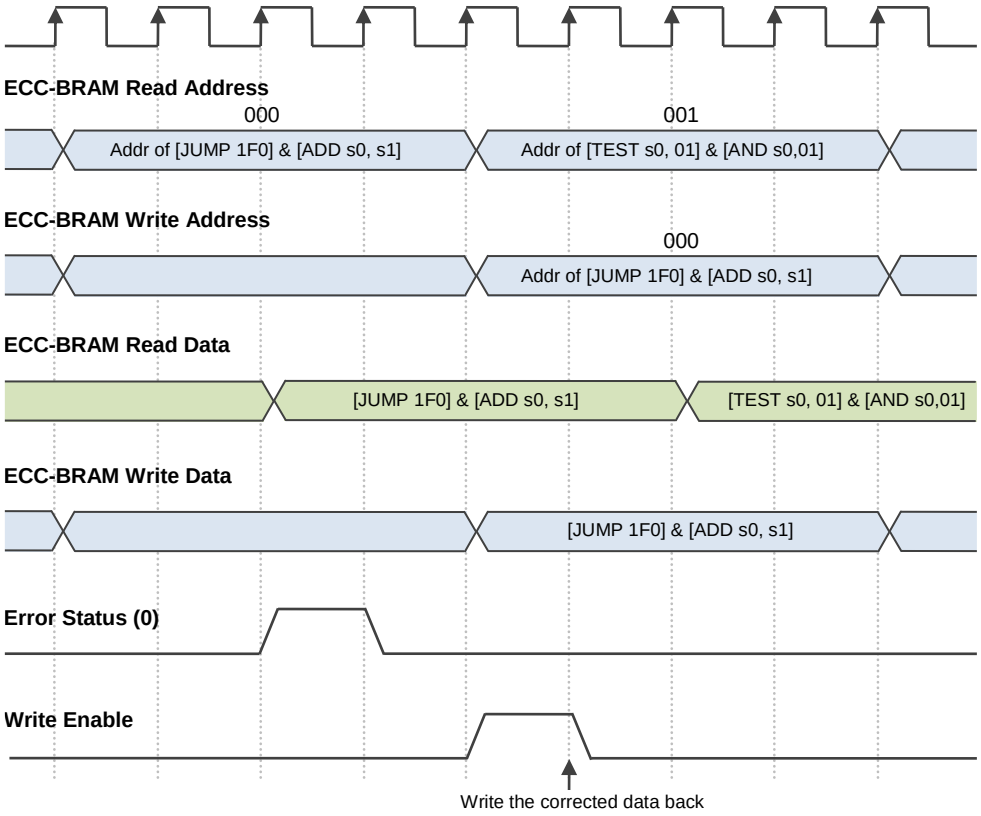


Figure 5.7 ECC-BRAM self-correction

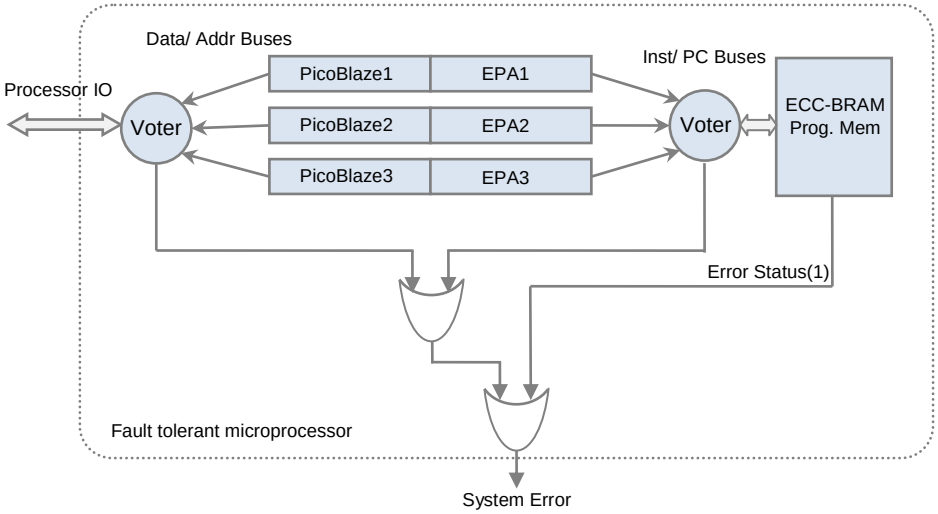


Figure 5.8 Fault tolerant microprocessor block digram

5.1.5 Bitstream Generation

Conventionally, ECC-BRAM is used as RAM rather than ROM, since the ECC code can only be generated by the ECC encoder when writing to the memory. Therefore, if ECC-BRAM is used for storing the pre-generated data content, the ECC code has to be generated separately in the design phase using off-line tools, and then merged with the original bitstream using the *data2mem* tool, which is a Xilinx design tool used for updating the bitstream with new BRAM data content [Xilinx2009b, Xilinx2011b]. Figure 5.9 shows the general steps used to update PicoBlaze's program memory with the user assembler code using the *data2mem* tool. First of all, the bitstream contains all of the hardware information, which is generated by synthesizing and implementing the original HDL files, such as Verilog or VHDL code. On the other hand, the updated content in the program memory, i.e. the machine code, is stored in a separate binary file, which is the executable hex file assembled by the PicoBlaze assembler from the original assembler code. In addition to the above, a memory map file is needed to specify the physical location of the machine code; namely, the address of the BRAM that is used to store the machine code. After the three files are generated, the *data2mem* tool is used to create a new bitstream, which replaces the BRAM's old content with the updated machine code.

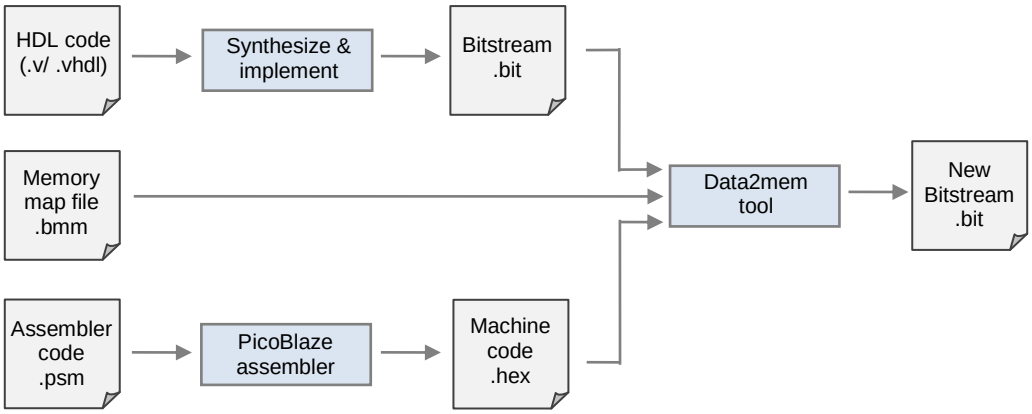


Figure 5.9 Steps to update PicoBlaze's program memory

When updating the content of the ECC-BRAM, not only does the data need to be updated, but also the ECC code has to be pre-calculated for the data. The ECC code can be calculated using a variety of off-line tools, such as *Visual Studio* or *Eclipse* running on a general processor. The methods used to calculate the ECC code and its implementation have been extensively studied [Hamming1950, Dietterich1995, and Yamauchi2000] and this is beyond the scope of this thesis. However, it is acknowledged that the software ECC code generator using off-line software tools has not been implemented yet, and in the current test version the ECC code is obtained by prewriting data to ECC-BRAMs on the FPGA and then reading back the bitstream, which includes both data and the ECC code.

5.1.6 Testing

The functionality of the EPA is tested by implementing different sets of instructions and comparing their outputs with those of the standard PicoBlaze. The results show that the fault tolerant microprocessor using ECC-BRAM produces the same results as the standard PicoBlaze. Furthermore, as part of the R3TOS project, the processor is tested by implementing the EDF scheduling and EAC allocation algorithms. Both algorithms deal with extremely complex matrix computations and a large number of iterations, whereby most of the instruction set is covered and tested. The results show that the result calculated by the proposed ECC-protected processor is all correct, with

less than 10% loss of speed, which results from the insertion of the NOP operations when the program branches [Hong2012a].

In order to verify the capability of recovering from a fault, the ICAP port is used to randomly inject faulty bits into the ECC-BRAM. Figure 5.10 gives the steps used to inject a fault, correct the fault and verify the correction. Note that the ECC-BRAM has to be disabled before it is accessed by the ICAP.

In terms of resource requirements, the ECC Processor Adaptor (EPA) is a lightweight hardware module that consumes only 46 slices when implemented on A Virtex-4 FPGA. It is also anticipated that there will be no significant increase in resource consumption when adapting it for more advanced soft-core processors such as MicroBlaze. On the other hand, the proposed ECC-protected processor (without TMR) occupies only 150 slices, and the whole fault-tolerant microprocessor (after implementing TMR and voters) requires less than 500 slices. This area requirement still represents a light footprint considering the large size of modern FPGA chips.

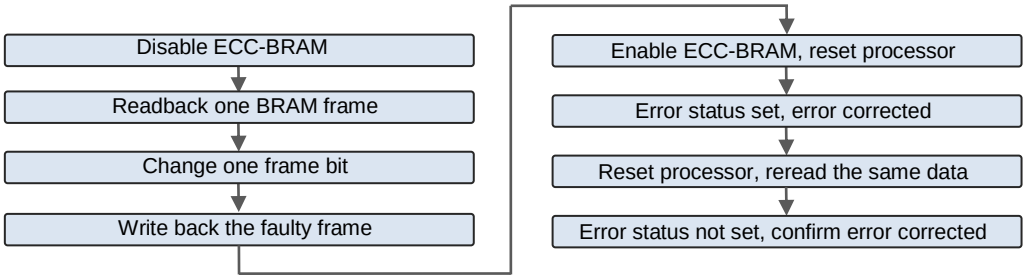


Figure 5.10 Steps to verify fault correction

5.2 Task Scheduler, Allocator and ICAP Manager

The task scheduler, allocator and ICAP manager are all implemented on the previously mentioned fault-tolerant microprocessor, and the scheduler and allocator are coded with the EDF scheduling and EAC allocation algorithms respectively. The algorithms have been presented in Chapter 4, and the hardware implementation of the microprocessor has been illustrated in Section 5.1. This section presents the technical details of their processing flows and memory managements when implemented and integrated with in the proposed system.

5.2.1 Task Scheduler

The task scheduler implements the EDF algorithm on the fault-tolerant microprocessor, whereby tasks can be reliably scheduled and executed according to their deadlines. The task scheduler acts as the master in the R3TOS HWμK, which communicates with the main CPU, the task allocator and the ICAP manager.

Algorithm implementation

Figure 5.11 presents the functionality of the scheduler. The scheduler runs infinitely in the iterations of scanning task queues. The program begins by scanning the finished task queue, which records the task IDs of all the finished tasks. If any task is in finished status, the scheduler will inform the allocator to update the FPGA state matrix, so that the cells occupied by the finished task are freed and marked as available in the FPGA state matrix. If the allocator is ready, the parameters of the finished task will be sent to it, including the task position and the task size. After receiving the parameters, the allocator updates the matrix, and returns the updated chip MER to the scheduler once it finishes the matrix computation. Meanwhile the scheduler changes the status of the task to be allocated. Note that in this stage, the finished task is only removed from the matrix rather than physically removed from the chip. By doing this, the finished task can still be reactivated without reconfiguration, once it is required by the main CPU again. On the other hand, since the task is removed from the matrix, the allocator can assume that these resources are available to be used by other tasks. The allocated task is physically removed only if a later coming task requires the same resources, in which case a blanking reconfiguration is performed. The blanking reconfiguration reconfigures the partial bitstream of a particular area to zero, before the new task is allocated. After blanking, the finished task is marked as removed, so that it has been physically de-allocated and requires reallocation if it is required again. If a finished task is required before it is removed, thus needing to be reactivated, it can easily be reactivated by changing its current status from allocated to ready.

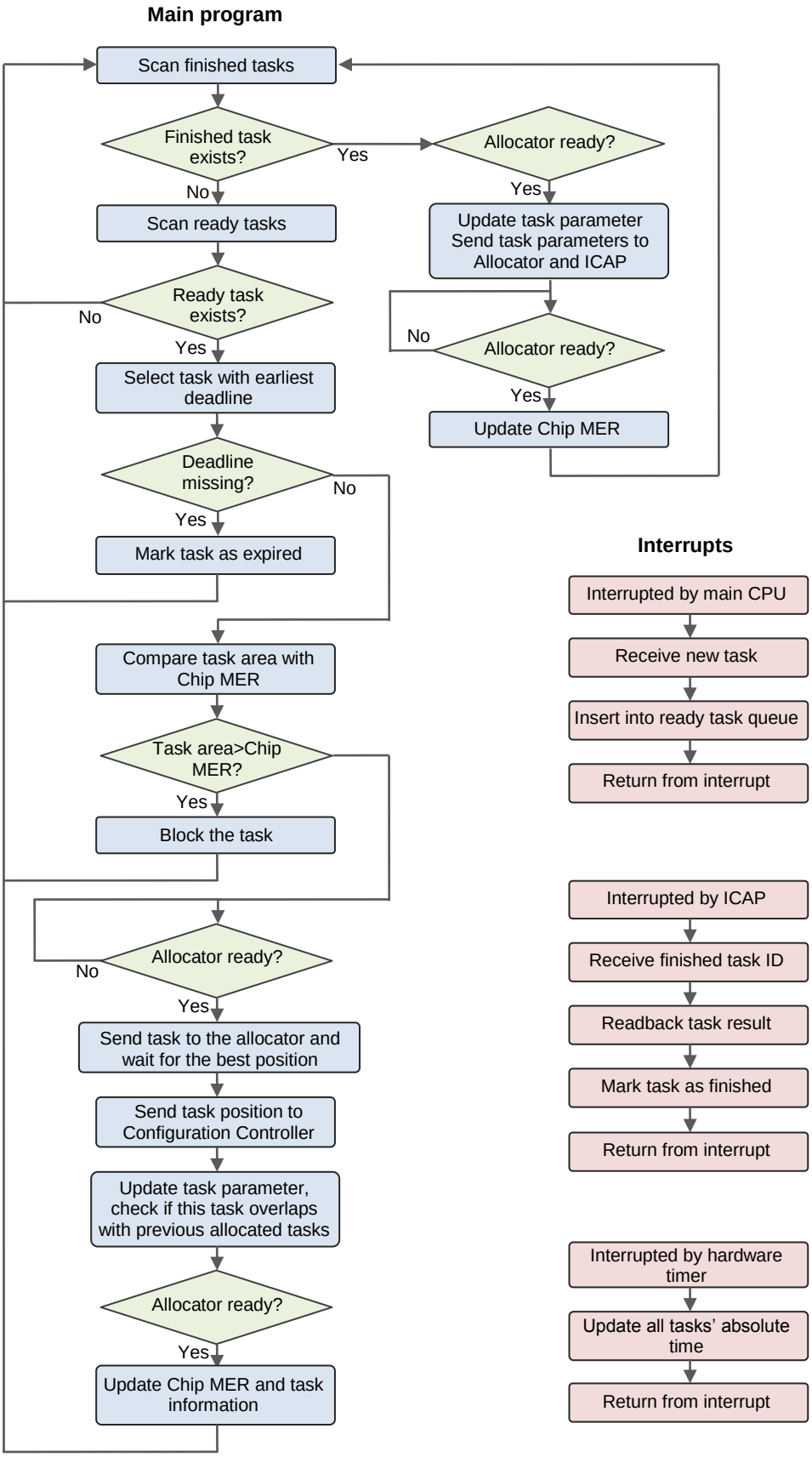


Figure 5.11 Task scheduler program flow

If there is no finished task, the scheduler will scan the ready task queue, in which all the waiting tasks are sorted by their absolute deadlines. The task at the head of the queue is scanned first since it has the earliest absolute deadline. If the deadline of the task has not passed, the size of the task will be compared with the chip MER; otherwise the task will be marked as expired. Tasks with sizes larger than the chip MER will be immediately blocked, and then the task with the next earliest deadline will be prioritised. If the task size can fit into the current chip MER, the task parameters will be sent to the allocator, including the task width and the task height. The task allocator searches for the best location for the received task, and then returns the best location if the task can fit into the currently available resources; otherwise, the task will be blocked. After the scheduler receives the position, it updates the task parameters with the new position and changes the status of the task to be allocated. In this stage, all the other allocated tasks, i.e. those which have finished their execution but not removed, are checked and compared with the new task. If the area occupied by the new task overlaps with any of the previously allocated tasks, the previously allocated task will be physically removed via blanking reconfiguration. Afterwards, the scheduler sends the task ID and the new position to the ICAP manager, which loads the bitstream from the bitstream library, updates the task position and then configures it to the chip. During the configuration time, the task allocator updates the matrix with the new FPGA status, and returns the updated chip MER to the scheduler.

The scheduler can be interrupted from three sources namely: the main CPU, the ICAP manager, and the hardware timer. The interrupt from the main CPU is used to insert a new task into the ready task list, which has the highest priority. To pass the task information from the main CPU to the scheduler, the task memory, which stores all the tasks and their information, is shared between the scheduler and the main CPU. To insert a new task, the main CPU needs to buffer the new task information to the shared memory, and raise the interrupt signal to the scheduler. The scheduler receives the new task and inserts it into the ready task queue according to its absolute deadline, before it returns to the main program. The interruption from the ICAP manager is used for notification that a task has finished its execution, which has the

second highest interruption priority. A task in R3TOS can have two types of execution time namely: the deterministic execution time, and non-deterministic execution time. The former is used for tasks whose finishing time can be predicated so that the scheduler can schedule them according to their deterministic execution time, whereas the latter cannot be predicted and therefore the scheduler has to wait for the finishing signal from the task. If a non-deterministic task finishes its execution, the ICAP manager will detect its stage, readback its results, and then interrupt the scheduler. The scheduler receives the task ID and its result, then marks the task as finished and inserts it into the finished task queue. The hardware timer interrupt is used to cope with the time overflow. Every time the hardware timer overflows, the scheduler will receive an interrupt, and then updates all the tasks' absolute time values.

Task BRAM mapping

The task scheduler uses a BRAM to store all information about tasks namely: the task BRAM. The task BRAM is configured as a memory block which can hold up to 2048 8-bit data (from 0x000 to 0x7FF). Figure 5.12 shows the memory map of the task BRAM. The memory can store up to 60 tasks at one time, and each task consumes 32 bytes of data, in which 29 bytes are used to store the tasks' parameters and 3 bytes are reserved without being used. The 29 bytes of task parameters include 3 bytes of basic information, 12 bytes of time information, and 14 bytes of area information. The basic information includes the task ID, the pointer to the next task in the queue, and the current status. The time information refers to all of the absolute and relative time data mentioned above in TABLE 4.1. The area information requests the parameters used by the allocator, including the task width, height, area, resource types, and absolute positions and its input/output data buffer. In addition, the task's critical level and the positions of its redundant modules, namely the FCCRs, are also included in the area information. The 60 tasks occupy the memory space from 0x000 to 0x77F, and the rest of the memory is used for storing the task list, the input/output buffer and the program stack. The ready task queue and the executing task list keep a copy of tasks in the ready state and the execution state respectively, which are used by the scheduling algorithms. The program stack is used to save the program

context, which is all of the register values, when a subprogram is called. The main CPU IDB and ODB are the memory spaces shared by both the scheduler and the main CPU for data exchange. For instance, to send the task input data to the scheduler, the main CPU firstly buffers the data to the main CPU's ODB in the task BRAM, and then sends an interrupt to the scheduler. The scheduler receives the interrupt and then configures the input data to the task's ODB. Likewise, the output from the task can be temporarily stored in the main CPU's IDB, and the main CPU will receive an interrupt from the scheduler to read the data from the task BRAM to the host.

5.2.2 Task Allocator

Allocator is coded by the EAC allocation algorithm, and implemented on the fault-tolerant microprocessor.

Main functioning

The task allocator communicates with the task scheduler and provides an optimized resource management service to the scheduler. Figure 5.13 shows the main program of the allocator. The allocator firstly waits for new task information from the scheduler by polling the new task signal. If a new task is requested from the scheduler, the allocator will receive the area information and the command, such as allocation or de-allocation; otherwise, it keeps polling until the scheduler makes a request. If it is an allocation command, the allocator will search for the best position using Algorithm 4.6, then return the best position to the scheduler once the best position is found; otherwise, a reject signal will be returned to the scheduler. After the position is decided, the allocator updates all the matrics using Algorithm 4.3, Algorithm 4.4, and Algorithm 4.5. Finally, the updated chip MER is returned to the scheduler to be used for scheduling later coming tasks. De-allocation is similar to allocation, except that no position result is needed, and the allocator only needs to update the matrices with the removed task, and return the chip MER to the scheduler.

The task allocator is responsible for searching for an optimum position to accommodate tasks, whereby better resource usage efficiency can be achieved. The

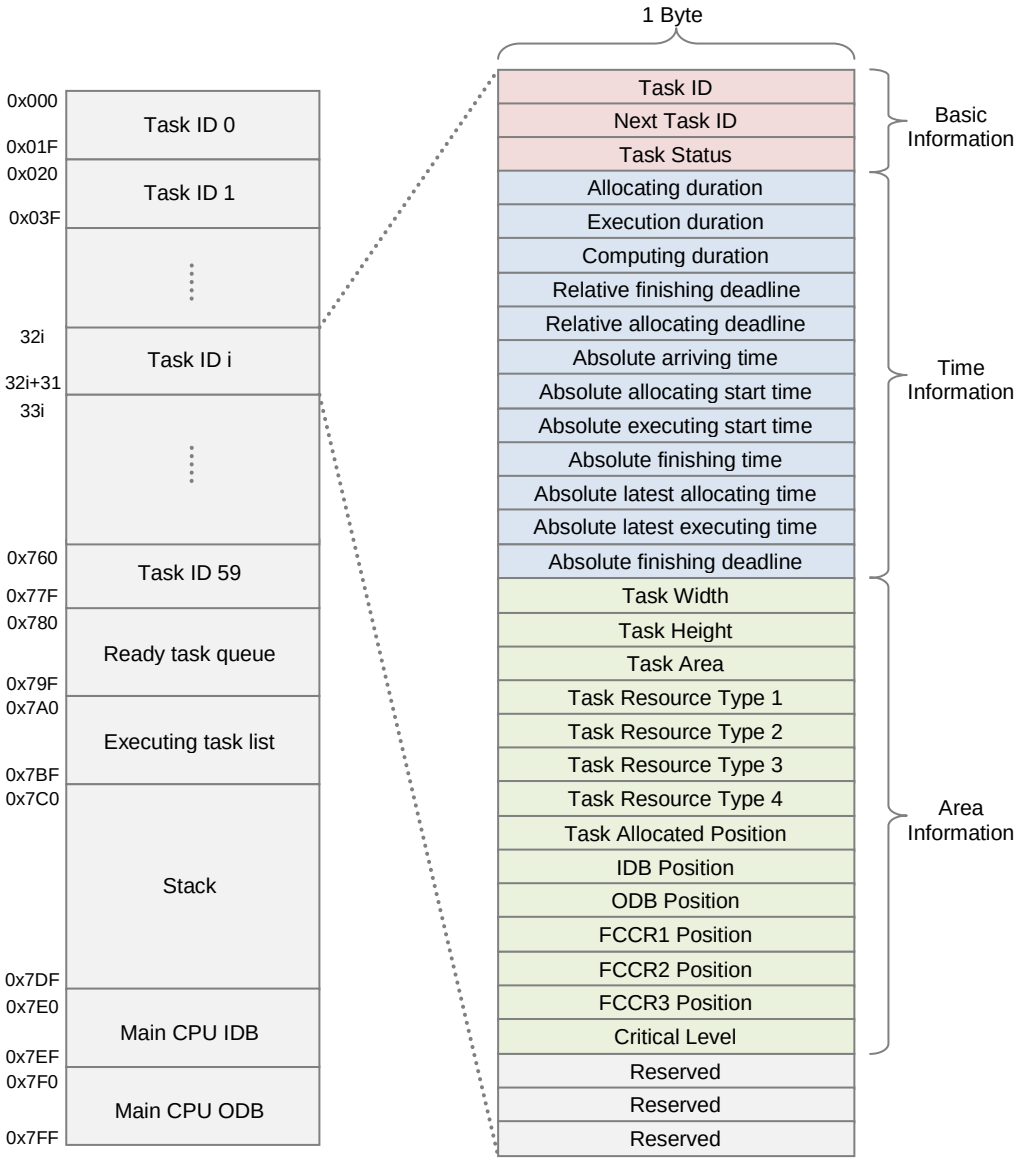


Figure 5.12 Task BRAM memory mapping

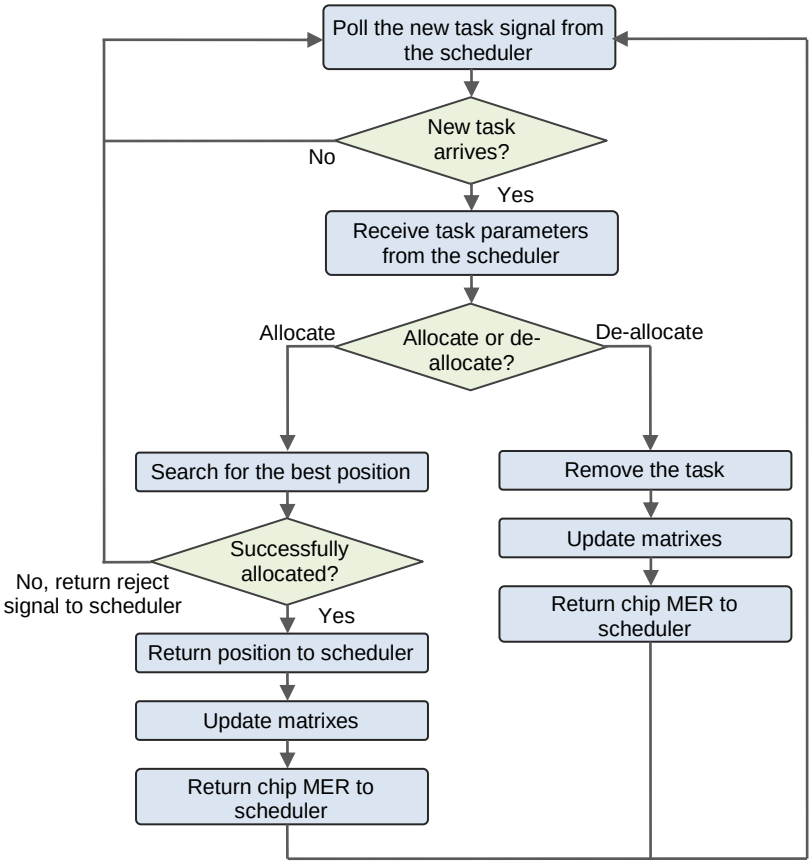


Figure 5.13 Allocator main program flow

Matrix BRAM memory mapping

Similar to the scheduler, the allocator uses a matrix BRAM to store all of the chip area information in terms of state matrices. The matrix BRAM can store up to 2048 bytes of data, which are used for eight matrices and the program stack (see Figure 5.14). In the current version of the system, each matrix can support up to 16 × 12 cells and each cell is represented by an 8-bit value. In the FPGA state matrix, the value of cells represents the status of CLBs, with values of 0x00 for free CLBs, 0x01 for CLBs occupied by tasks, and 0x10 indicating that it is a damaged resource. In the other matrices, the cell values are used for calculation, ranging from 0 to 255 in decimals. Note that the current matrix BRAM, as does the task BRAM, uses the standard BRAM block, rather than ECC-BRAM. When adapting to ECC-BRAM, the memory space is doubled since the ECC-BRAM is composed of two standard

BRAMs. The doubled memory space can support larger sizes of matrices, and therefore can be used for larger chips or finer granularity.

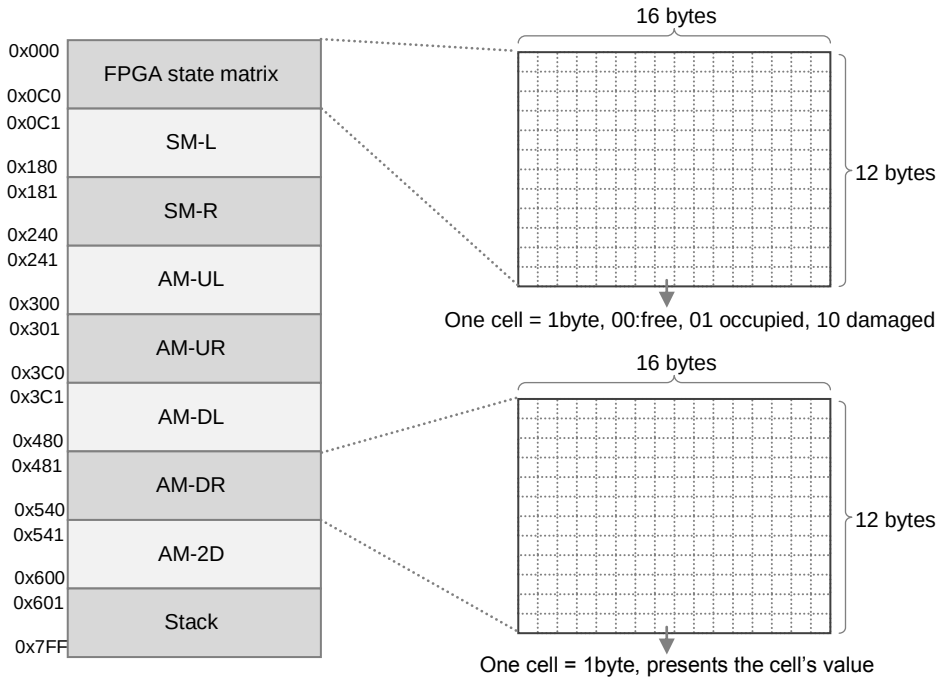


Figure 5.14 Matrix BRAM memory mapping

Resource Type Checker (RTC)

The Resource type checker (RTC) is a hardware module attached to the allocator used to differentiate between resource types. The RTC scans each column of resources in parallel with the allocator, and compares the task's resource with the chip's resource in a shifting mode. Figure 5.14 gives an example of checking if the resource type of the task matches with that of the chip. The chip resources are divided by vertically aligned columns, including IOB, BRAM, and CLB. When allocating a task, the task's position is shifted from right to left and its resource type is compared with the chip's resource type, and the position becomes valid only if the resource type matches. The chip resource type is loaded into the RTC during system initialization, and the task resource type is updated each time a new task is inserted. When searching for a possible position, the allocator scans all of the cells row by

row, from bottom to top; and in each row, cells are scanned from right to left, which is synchronised in parallel with the task shifting in the RTC. During scanning, the RTC outputs a valid signal of 1 bit to indicate if the resource type matches at the current position, and the allocator uses this signal to find the matched resource type.

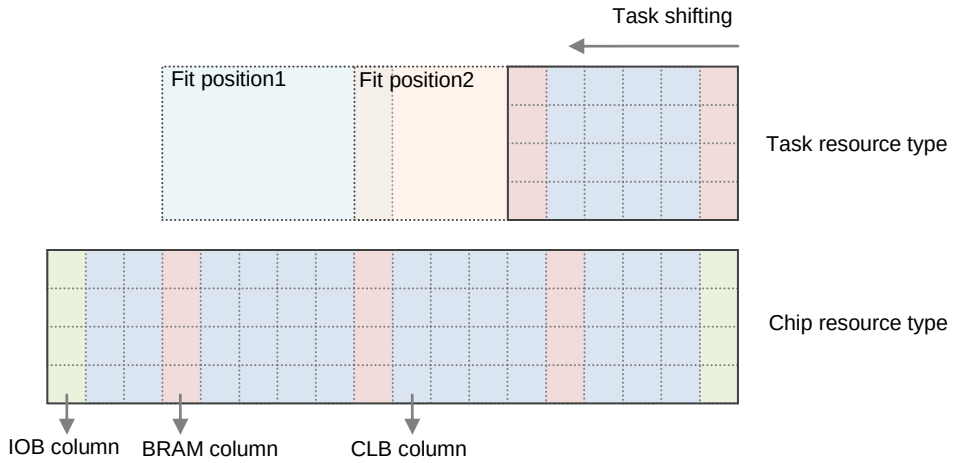


Figure 5.15 Column Shifting of RTC

Pipeline and communication

When scheduling and allocating a task, the three components in the HWμK, namely the task scheduler, task allocator, and ICAP manager, are pipelined to achieve better performance (see Figure 5.16). Here, stage1 and stage2 are sequentially programmed, during which operation the scheduler schedules tasks and the allocator searches for best positions. After the best position is decided by the allocator, all of the three components can start operating in parallel. During stage3, the task scheduler updates the status of tasks, updating the current task's status to executing, and checking whether it overlaps with previously allocated tasks, while the allocator is updating matrices with the new task position and the ICAP manager can start loading the bitstream into the configuration memory.

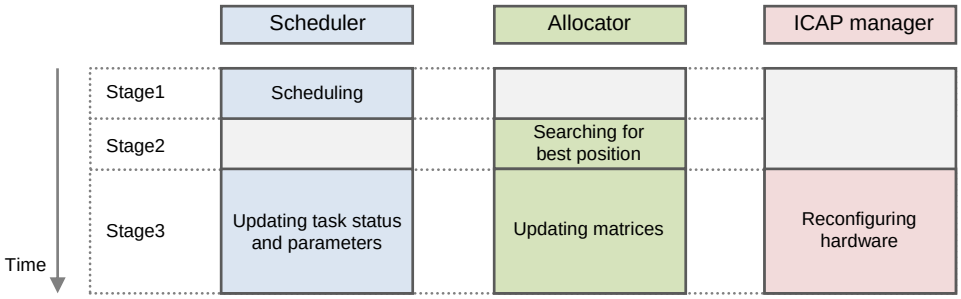


Figure 5.16 Three processes pipeline

The task scheduler, task allocator and ICAP manager are connected together using bus-like internal links. The scheduler controls the links by giving a set of commands to control both the allocator and the ICAP manager. TABLE 5.2 shows the scheduler's command code used for communication between the allocator and the ICAP manager. The 2-bit address is used to multiplex between the allocator and the ICAP manager, while the 8-bit instruction indicates the type of the 8-bit data [Hong2011b].

TABLE 5.2 COMMUNICATION CODES IN HWμK

Instruction hex code	Commands to Allocator Address: "01"	Commands to ICAP manager Address: "10"
0x00	Idle	Idle
0x01	Insert/Remove Task	Insert/Remove Task
0x02-0x06	Sending Task Parameters	Sending Task Location
0x07	Searching Best Location	Send Bitstream Address
0x08	Updating Matrixes	Reserved
0x09	Feedback Chip MER	Reserved
0x0a-0xff	Reserved	Reserved

5.2.3 ICAP manager

The ICAP manager directly controls the bitstream and the configuration memory, providing reconfiguration facilities of the lowest level hardware. It mainly consists of the previously proposed fault-tolerant microprocessor, a bitstream BRAM and a hardware ICAP driver.

Main functioning

The ICAP manager provides a set of low level services, including inserting and removing a task, scrubbing, and writing and reading a single frame. Figure 5.17 shows the general functioning flow of the ICAP manager. The program starts by polling the operation code from the scheduler, then branches to different subprograms. When inserting a task, the ICAP manager firstly receives the task ID and the task position as updated by the allocator, then uses the ID to load the bitstream from the external bitstream library and to change its Frame Address Register (FAR) to the updated position. The FAR is the address of the first frame in the bitstream, which is used to define the position of a partial bitstream [Xilinx2009]. After updating the FAR with the new position, the bitstream is loaded into the FPGA's configuration memory. When finishing, the ICAP manager sends a signal to notify the scheduler of the job completion. To remove a task, the task area and position are used, whereby the ICAP manager can blank a particular region on the FPGA by configuring all-zero frames to the target area. The single frame reading and writing is similar to task reading and writing, whereas the frame is buffered in a temporary on-chip memory, i.e. a BRAM, rather than the external memory. The ICAP manager is also capable of scrubbing the whole configuration memory to improve the chip's fault tolerance. When a scrubbing request is sent from the scheduler, the ICAP manager firstly restores the last scrubbed frame address, and then continues reading back a certain number of frames, during which the frame ECC facility is used to check the frame correctness. When finished, the latest scrubbed frame address is stored, which will be used for the next round of scrubbing.

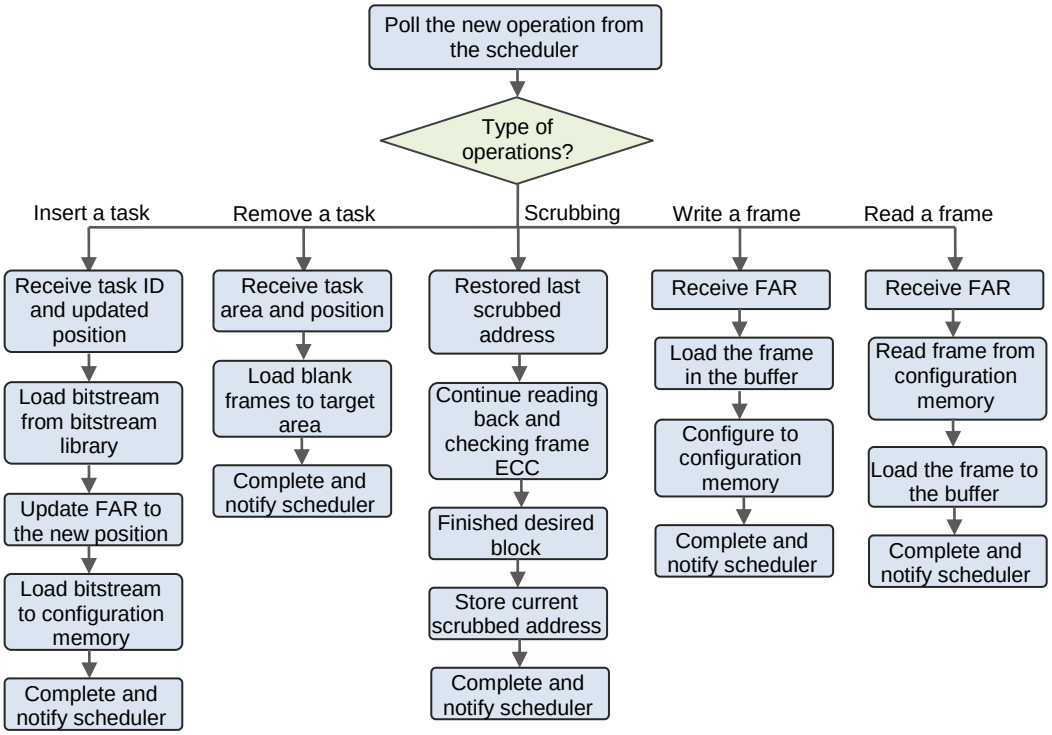


Figure 5.17 ICAP manager functioning flow

Bitstream BRAM memory mapping

The bitstream BRAM is used to store bitstream commands and temporary frames. As with the other BRAMs, the bitstream BRAM can store up to 16,384 bits; however, the data width is set to 32-bits instead of the previously used 8-bit data. The 32-bit data width is compatible with the frame word and bitstream commands, which are all 32-bit. Figure 5.18 depicts the memory organisation of the bitstream BRAM, in which a set of command sequences are stored, including the command sequences of initialisation, single frame writing/reading, area (task) writing/reading, and multiple frame writing (MFW). Each sequence is allocated with 32 data spaces (128 bytes) and 256 data spaces (1024 bytes) are reserved for the frame buffer, which can buffer up to 6 frames at one time.

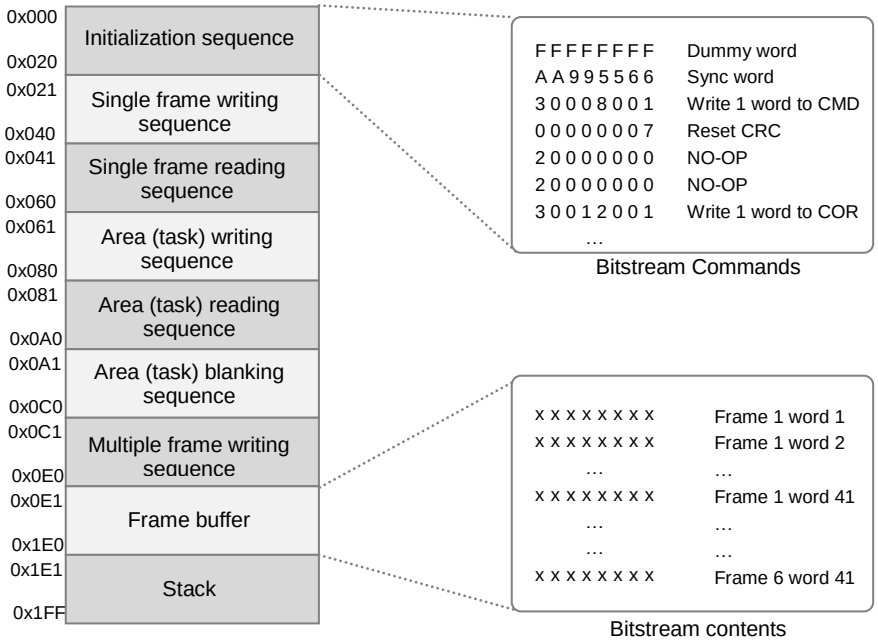


Figure 5.18 Bitstream BRAM memory mapping

Hardware ICAP driver

The ICAP driver is a hardware state machine that achieves the maximized throughput of the current ICAP. Figure 5.19 shows the connections of the ICAP driver when integrated with the ICAP manager. The main function of the driver is to load the bitstreams from the frame buffer (the bitstream BRAM) to the configuration memory and vice versa. The driver is implemented on the ICAP, which has a 32-bit data width running up to 100MHz in Vertex-4 family FPGAs. The driver can load data from/to the ICAP port in every clock cycle, while automatically toggling control bits such as the enable bit. In addition, the driver can cope with low level hardware complexes. For instance, the ICAP requires a pad frame and a pad word before writing a frame and a word respectively, and the bitstream in the bottom half of the FPGA is reversed from the top half. To cope with this problem, the ICAP driver can pad dummy frames/ words, and reverse bits in a single word, as well as words in a single

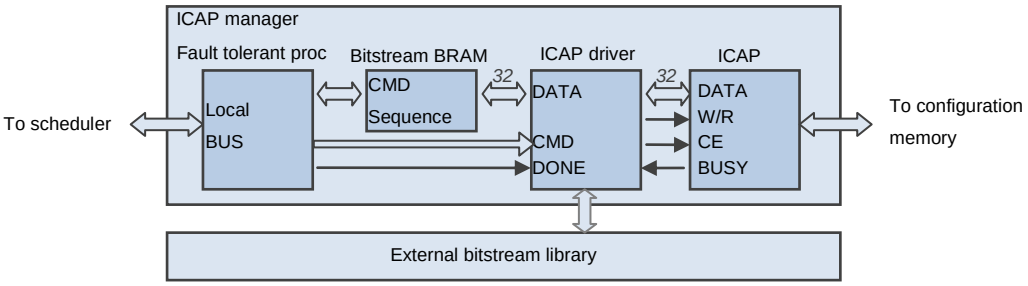


Figure 5.19 ICAP manager

frame. In addition, the ICAP manager is fault-tolerant, and thus it is capable of recovering from errors. The detailed implementation is not included in this thesis since it has been published by the research group member Ali Ebrahim [Ebrahim2012], and a description of its integration into the system has also been published [Iturbe2013c].

5.3 SMP Communication Architecture

The R3TOS supports four types of inter-task communication, including ICAP based communication, Data Relocate Task (DRT)-based communication, symmetric multiprocessing (SMP)-based architectures, and the snake strategy. The four communication mechanisms are designed for different application demands. ICAP-based communication requires less area overhead but provides limited bandwidth, and therefore is used for Low-Bandwidth Communication (LBC) tasks [Iturbe2011b]. Whereas DRT-based communication achieves unlimited communication bandwidth at the cost of large logic resources, and hence is suitable for High-Bandwidth Communication (HBC) tasks [Iturbe2013c]. SMP-based architecture gives a balanced communication throughput by trading off its area overhead, and its shared memory programming interface is particularly interesting for SMP-like applications [Hong2012b]. The snake strategy physically concatenates tasks to achieve maximum throughput with a smaller footprint, whereas tasks have to be designed under particular constraints [Iturbe2011a]. The communication methodologies have been introduced in general in Chapter 3, and this chapter mainly focuses on the implementation of SMP-based communication. The other three

communication mechanisms have been implemented by group member Xabier Iturbe, and the detailed implementation schematics have been published [Iturbe2011a (snake strategy), Iturbe2011b (ICAP), and Iturbe2013c (DRT)].

The proposed SMP communication architecture is inspired by a shared memory programming interface, namely OpenMP, in the multiprocessor programming paradigm [Chandra2000]. One of the similar features of FPGAs and multicores is their task parallelism. In multicore approaches, software tasks are partitioned into multiple threads running parallel in multiple cores, while hardware tasks in FPGAs are separately implemented on dedicated circuits operating independently at the same time. Motivated by above, a hardware virtual SMP architecture was designed and implemented, which provides an OpenMP-like programming interface in the hardware scenario [Hong2012b]. In this architecture, the content of the physically separated BRAMs is shared virtually among all tasks, whereby hardware tasks can communicate with each other using the same content in the shared memory. To ensure the coherence of the contents of different BRAMs, the ICAP is used to synchronise the memory every time data is updated. In addition, to reduce the communication overhead, the compressed bitstream writing mode is used to broadcast the same content to multiple BRAMs. The results show that synchronisation speed is significantly improved at low area cost.

5.3.1 SMP Architecture in Software

In the software SMP architecture, multiple processors can have a shared memory space where a single address space can be used across the whole memory system, whereby inter-processor communication can be achieved by exchanging data in this shared memory space [Chandra2000]. Figure 5.20 shows the general software SMP architecture using a shared memory space. Here, the private data can only be accessed by the owning thread (or processor), whereas the shared data can be accessed by all processors for inter-processor communication. A network is used to provide access to the physical memory for data ordering and multiplexing.

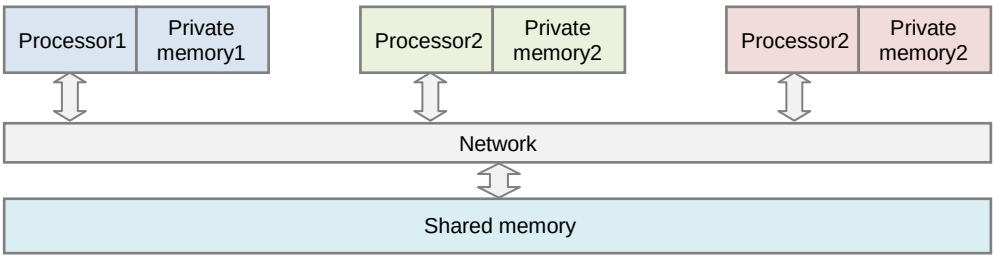


Figure 5.20 SMP architecture using shared memory space

Figure 5.21 gives an example of inter-processor communication using the shared memory. In this example, the first process (or thread) passes its private data to the second process (or thread) by copying its private data to the shared memory and the second processor can access the shared data by reading from the shared memory.

Based on the SMP architecture and shared memory-based communication, the OpenMP provides a standard programming interface for multicore applications. An example of OpenMP programming is given in Figure 5.22. In order to be compiled as an OpenMP program, the program starts with an OpenMP directive at the first line, which includes the OpenMP sentinel (#pragma omp). The main program begins its execution on a single thread, which is also called the master thread. A team of threads will be created by the master thread when a parallel region is encountered. The parallel region is the basic parallel construct in OpenMP, in which all of the team threads run in parallel. When the parallel region reaches the end, the master

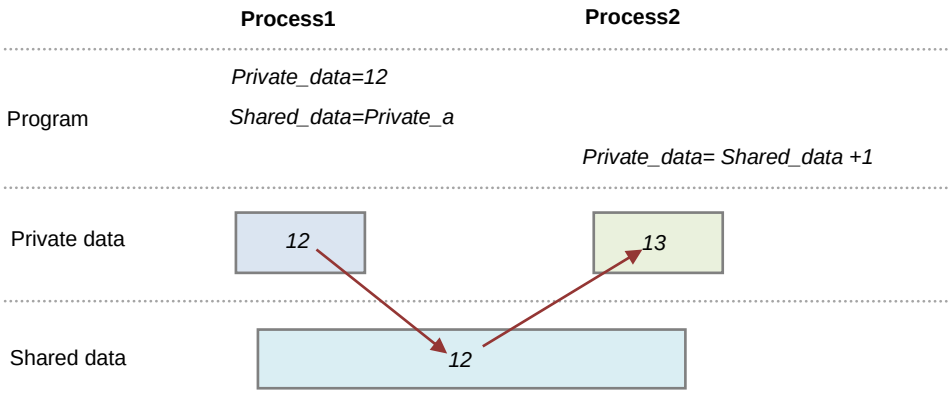


Figure 5.21 Data communication using shared memory

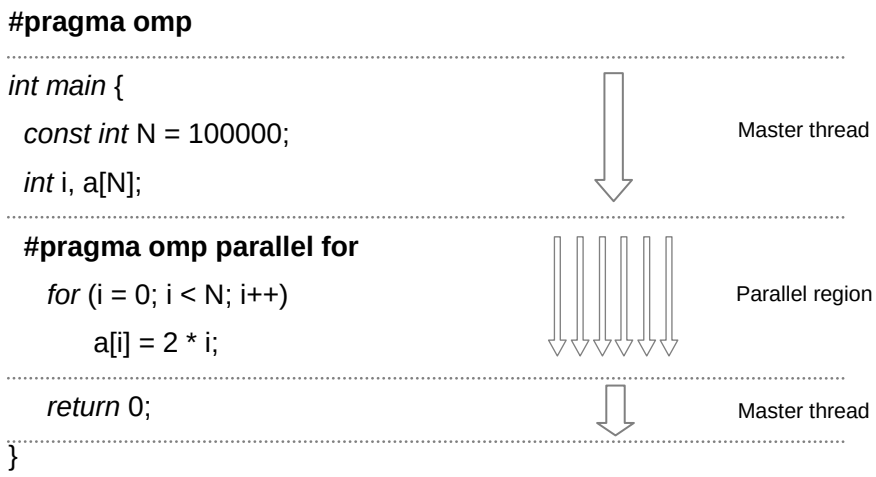


Figure 5.22 OpenMP programing example

thread will wait until all of the parallel threads finish their computations, before continuing to its next statement. Note that the parallel region is automatically partitioned and parallelised by the compiler, according to the program independency and the number of cores, and communication between cores is realised by using a shared memory.

5.3.2 Virtual SMP Architecture in Hardware

In the proposed virtual hardware SMP architecture, all of the hardware tasks have at least one BRAM. Although the BRAMs are physically separated around the whole chip, they can hold the same contents all the time. If any of the BRAMs is changed by its owning task, all of the other BRAMs will be synchronised with the same updated content. Therefore, a virtual shared memory is implemented for inter-task data exchange and communication. Content synchronisation is achieved using the ICAP rather than conventional routing, so that once a BRAM's content is changed, the ICAP manager will broadcast the updated content to all of the other BRAMs.

The hardware framework has been implemented and enables shared memory programing. Figure 5.23 shows an example of the hardware SMP architecture implemented on a modern FPGA chip. The chip is divided into vertically aligned clock regions, and most often every clock region consists of an identical number of resources. In each clock region, the CLBs are homogeneously distributed, and a

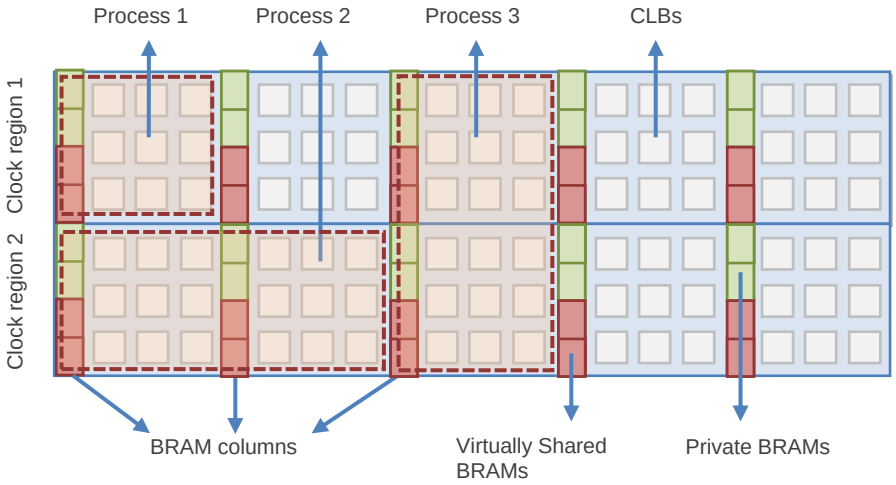


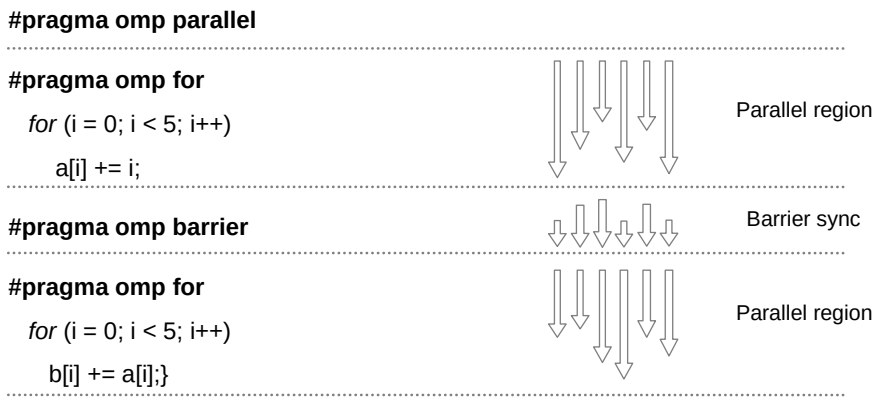
Figure 5.23 Hardware shared memory implementation

variety of different types of resource blocks are embedded for specific types of usage. The most commonly used specific block is the BRAM block, which provides compact memory with high storage density. The BRAMs are distributed within columns horizontally aligned in each clock region (see Figure 5.23). In the present implementation, task data and synchronisation are based on BRAM columns, as hence each task has to cover at least one BRAM column for data exchange. The BRAM column usually consists of four BRAM blocks. In this context, the shared memory is implemented on the lower two BRAM blocks, while the upper two are used for private memory. All of the shared memories keep the same content all the time for synchronisation and data exchange. In particular, the ICAP manager is used to keep all shared BRAMs coherent, instead of using the routing logic. Therefore the area overhead is reduced and chip fragmentation is avoided, which is in line with the context of the R3TOS, where tasks can be arbitrarily placed anywhere on the chip.

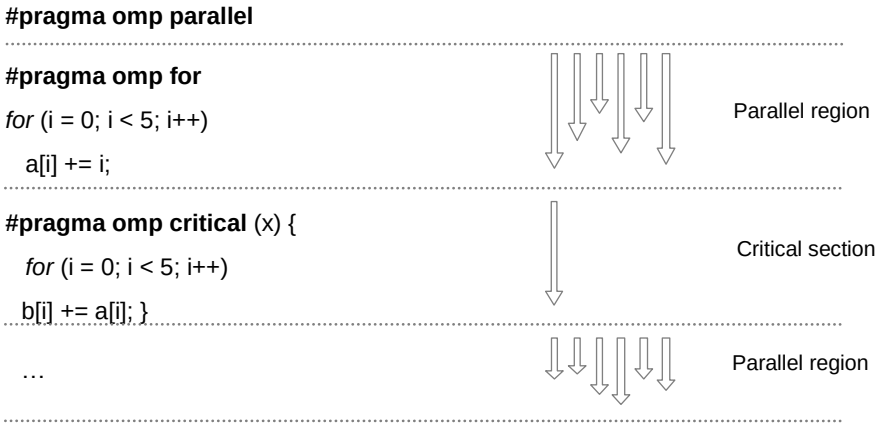
In order to synchronise the content, the ICAP manager performs a readback and broadcast operation. When a task writes new data to the shared memory, it can be detected by the ICAP manager. Afterwards, the ICAP manager firstly reads back all the frames which contain the content of the shared memory, then configures the frames to all the shared memory BRAMs. To reduce the time overhead in updating all of the BRAMs, the compressed bitstream is used for fast configuration, which

improves configuration by up to 10x [Hong2012b]. A detailed time analysis is demonstrated in the context of applications presented in chapter 6.

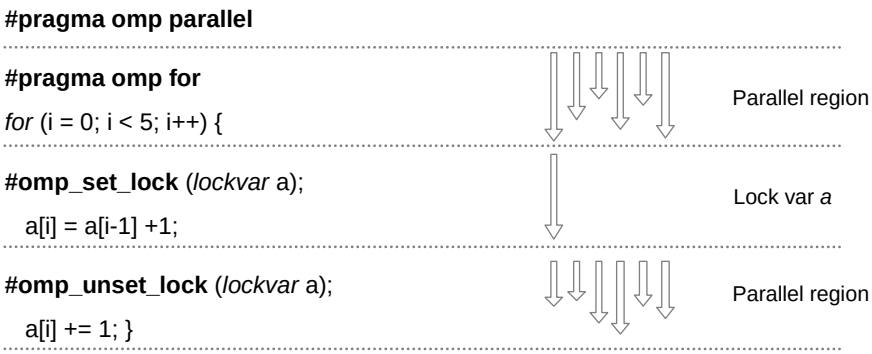
To provide support for the standard OpenMP-like programming environment, as well as to facilitate inter-task synchronisations, the traditional SMP programming directives have been implemented, which include program barriers, the critical section, and the lock variable. The barrier directive can be used to synchronise all the tasks at a particular stage of a program's execution, so that no task can proceed past a barrier until all of the tasks have arrived (Figure 5.24.a). In the standard OpenMP, the critical section refers to one block of program code that can be executed by only one thread at any one time (Figure 5.24.b); whereas in the proposed hardware scenario, the critical section is used by one hardware task to stop the execution of all other tasks, so that if one of the tasks goes into its critical section, all other tasks will be paused until the task finishes its critical section. The critical section directive can be used to protect updates to shared data. The lock directive is used to prevent all other tasks from accessing a particular item of data, where locked data can only be accessed exclusively by a single task, and other tasks trying to access it have to wait until the data is unlocked (Figure 5.24.c). In the proposed hardware SMP architecture, the three directives are implemented on three flags and mapped with three Look-Up-Tables (LUTs) respectively (see Figure 5.25). The LUT is a memory-like register array in an FPGA, which is the primary element of a CLB. One LUT can be configured as a distributed RAM, ROM, or be used as multiplexers for different logic functionalities. In Virtex-4 family FPGAs, one LUT consists of 16 bits, which can be assigned to 16 flags. As a consequence, the proposed implementation of three LUT can support up to 16 barrier flags, 16 critical section flags, and 16 lock flags. The three LUTs are stored in the first CLB column of a task and all of the flags are supervised and mastered by the ICAP manager. The contents of LUTs are accessible by both the task logic and the ICAP manager, where the ICAP manager can access the content of LUTS by writing/reading bitstreams to/from the configuration memory, while the task logic can access the content of LUTs using the conventional memory write/ read operation. When a task tries to use a program directive, it needs to raise a request by modifying the content of its flag-LUTs, and then waits for



a) Barrier directive



b) Critical section directive



c) Lock directive

Figure 5.24 Shared memory programing directives

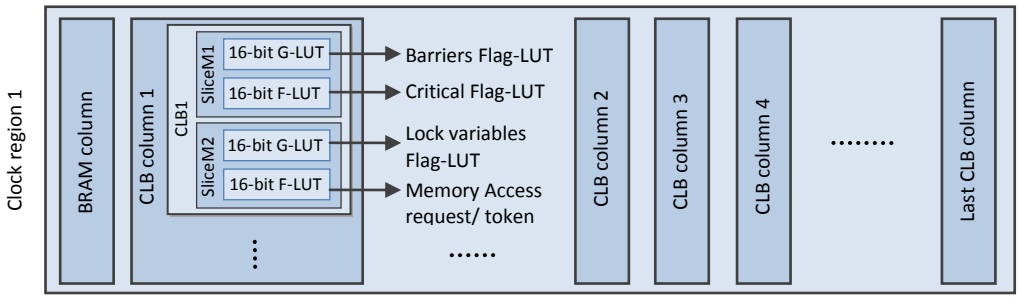


Figure 5.25 Implementation of directives

acknowledgement from the ICAP manager, which periodically polls all the tasks' flags.

In addition, in order to ensure the coherence of all of the shared memories, a hardware token is implemented to guarantee that only one task can access the shared memory at any one time. A task has to raise a request bit and wait for the token to be granted by the ICAP manager before it can change the content of the shared memory. The content of the shared memory can only be changed by tasks granted with the token. Similar to the other flags, the token is implemented on one LUT in the first CLB column, which is also polled by the ICAP manager periodically (see Figure 5.25). The frequency of the polling varies with the number of currently executing tasks. In order to minimise the overhead of the communication protocols, frequently writing to the shared memory is not advisable and tasks are recommended to write larger data blocks at one time once the token is authorized to them.

5.3.3 Results

The virtual SMP hardware has been implemented on a Xilinx XC4VFX60 FPGA. The implementation is tested separately from the other components in the R3TOS (task scheduler and allocator); therefore, it gives an independent evaluation of the architecture itself. The single virtual SMP architecture implementation includes only the ICAP manager and the ICAP driver. TABLE 3.1 gives the resource requirement breakdown on a Xilinx XC4VFX60 FPGA. The results show that the ICAP driver consumes 357 slices while the ICAP manager requires 96 slices (for the PicoBlaze core) and 3 BRAMs (for the program memory and the bitstream BRAM)

[Hong2012b]. Note that the result is evaluated before it is triplicated by the TMR. The total resource consumption without TMR is 453 slices and 4 BRAMs, which only accounts for 1.7% and 1.6% respectively of the total slices and BRAMs on the chip. The results for speed performance are presented in

TABLE 5.4, which gives the time required to perform different operations. When working at 100 MHz frequency, reading back all the contents of one BRAM column requires 26.81 μs, which is the same time taken to configure one BRAM column. The time needed to configure one BRAM column decreases when more columns are to be configured at the same time, by benefiting from the compressed bitstream which can be used to duplicate identical contents to multiple frames at the same time. To access the flags and the tokens, only one frame needs to be accessed, and hence this requires only 1.14 μs. It is acknowledged that the architecture has not been integrated into the R3TOS system, and this will be carried out in future work.

TABLE 5.3 SINGLE SMP CONTROLLER RESOURCE BREAK-DOWN ON A XC4VFX60 FPGA

Component	Slices	BRAMs
ICAP driver	357	0
ICAP manager (PicoBlaze)	96	3
Total	453	4
Percentage of Virtex4 FX60	1.7%	1.6%

TABLE 5.4 PERFORMANCE OF PROPOSED SMP ARCHITECTURE AT 100MHZ

Process	Cycles	Time
Readback from 1 BRAM Column	2681	26.81 μs
Configure to 1 BRAM Column	2681	26.81 μs
Broadcast to 10 BRAM Column	3918	39.18 μs
Broadcast to all BRAM Columns (58)	9294	92.94 μs
Read flags	114	1.14 μs
Release token	114	1.14 μs

5.4 Conclusion

This chapter presented the implementation of the system's low level hardware. The system consists of a hardware microkernel (HWμK) as well as a main CPU to provide a host API for user applications. The HWμK is mainly composed of three

kernels, namely: the task scheduler, the task allocator, and the ICAP manager. These are coded in the assembler and implemented on three fault-tolerant ECC-protected microprocessors.

This chapter presents the detailed development of the fault-tolerant microprocessor, including the design and implementation of the EPA, LookAhead strategy, and the self-recovery mechanism. Based on this, the implementation of the HWμK was then illustrated, in which both the program flow and the memory arrangement of the three kernels were detailed. Last, but not least, the design and implementation of the virtual hardware-based SMP mechanism was presented, which enables the communication between hardware tasks in the R3TOS. The implementation of the host API and system integration are not included since this work was conducted by group member Xabier Iturbe, and the integrated schematic level implementation has been published elsewhere [Iturbe2013c]. It is acknowledged that to date, the whole system implementation has not been fully integrated, and therefore the applications presented in Chapter 6 are demonstrated using part of the system. Future work will include fully integrating the whole system, so as to provide standard interfaces for all applications.

Chapter 6

Development of R3TOS Applications

The R3TOS is demonstrated in the context of two applications; namely, the K-Nearest Neighbour classifier (K-NN) and Sequence Alignment (SA). The K-NN is a classification algorithm used in data mining applications to calculate the distance between a query sample and members of a training set. SA is an application to demonstrate that a system is capable of parameterising tasks at run time, where the pipeline stage of a task can be adjusted according to currently available resources. This chapter presents the design and implementation of the two applications, in order to give a more detailed explanation of the system's operation through case-studies, as well as demonstrating its capabilities in task customisation, allocation, and fault tolerance, whereby resources can be more efficiently and reliably used.

The applications are developed in cooperation with the other members of the research group; namely, Hanaa Hussain, Mohammad Nazrin Isa, who have given their support in designing the standard processing elements for K-NN and SA respectively [Hussain2012a, Isa2012], while the author of the present thesis has customised their designs and integrated them with the system [Hong2013a, Hong2013b]. The subsequent two sections present two applications discussing the

Related publications: [Hong2013a] and [Hong2013b].

application's background, low-level hardware implementation, integration with the system and test results.

6.1 Case Study 1 – K-Nearest Neighbour Classifier

The classification of a high dimensional microarray database consumes a large amount of time when implemented on sequential programing-based GPPs. Therefore it has become preferable to implement them on FPGAs, which gives flexible processing parallelism and high throughput. Moreover, when applied with a DPR technique, not only can the multi-user multi-tasking environments be supported, but also the efficiency of resource usage and computing reliability are improved. The following sections first give an introduction to the K-NN algorithm, and then present the hardware implementation of a single K-NN core. Afterwards, the core is integrated with the R3TOS system and tested in the context of the proposed system.

6.1.1 The Non-parametric Classification Algorithm K-NN

Nowadays, microarray analysis has provided significant support in identifying gene patterns, such as the gene types of cancers or other diseases [Golub1999, Thijs200 and LeCao2009]. Faster and more accurate analysis can help scientists in many fields such as disease diagnosis, drug detection, and treatment personalisation, as well as outcome prediction [Golub1999, Macgrefor2002, and Brennan2005]. However, the databases used in such systems are not only extremely large but also highly dimensional, which leads to difficulties in data analysis. Therefore, data mining techniques are used to extract useful information before it is used in real applications.

In order to predict the class label of a set of samples, supervised classification is widely used, which identifies an unknown functional group by learning from known samples called training sets. One typical method is K-Nearest Neighbour (K-NN) classification, which is a non-parametric algorithm widely used in a number of bioinformatics applications. For example, K-NN can be used in post-genome data processing to discover new types of classes, such as defining a new cancer subtype, or to recognise class patterns that have already been discovered, such as when classifying an unknown sample to discovered known class labels [Golub1999,

Macgrefor2002, and Brennan2005]. In order to classify an unknown class label, the K-NN classifier firstly computes the distance between the query vector and all the training vectors, then identifies K vectors in the training set which have the nearest distance to the query. The query consists of M training samples, and each training sample can be stored in one B-bit data. The training set contains N training vectors, and each vector has M training samples, which are stored in the same way as the query. Figure 6.1 illustrates the vectors and matrices used in the K-NN algorithm. Here, a query containing M samples (Q_1 to Q_M) is compared with N training vectors. The N training vectors form an M×N matrix in which M is the number of samples in each vector, and N is the number of vectors. After computing the distances between the query and each training vector, the K (K=3 in this example) vectors with the nearest distances are selected to be the nearest neighbours.

The distance can be calculated using a variety of distance metrics, such as the Hamming distance, Euclidian distance, Manhattan distance, Cosine distance, Canberra distance, or others [Tahir2006, Manolakos2010]. In the present implementation, the Manhattan distance is used due to its simplicity and low-cost implementation compared with other distance measures. Given a query of Q= {Q_1, Q_2, Q_3, Q_4… Q_M} and a training vector of T_n ={$T_{n,1}$, $T_{n,2}$… $T_{n,M}$}, the Manhattan distance between Q and T_n can be calculated by Equation 6.1:

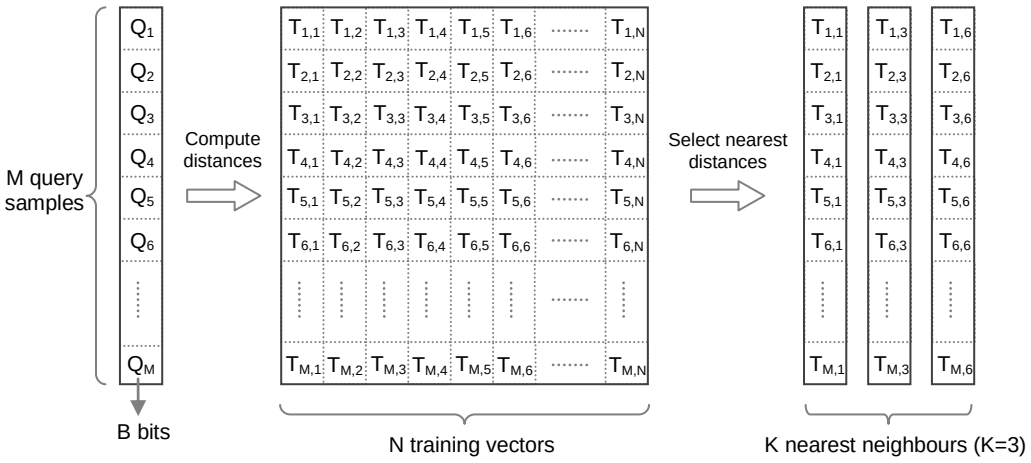
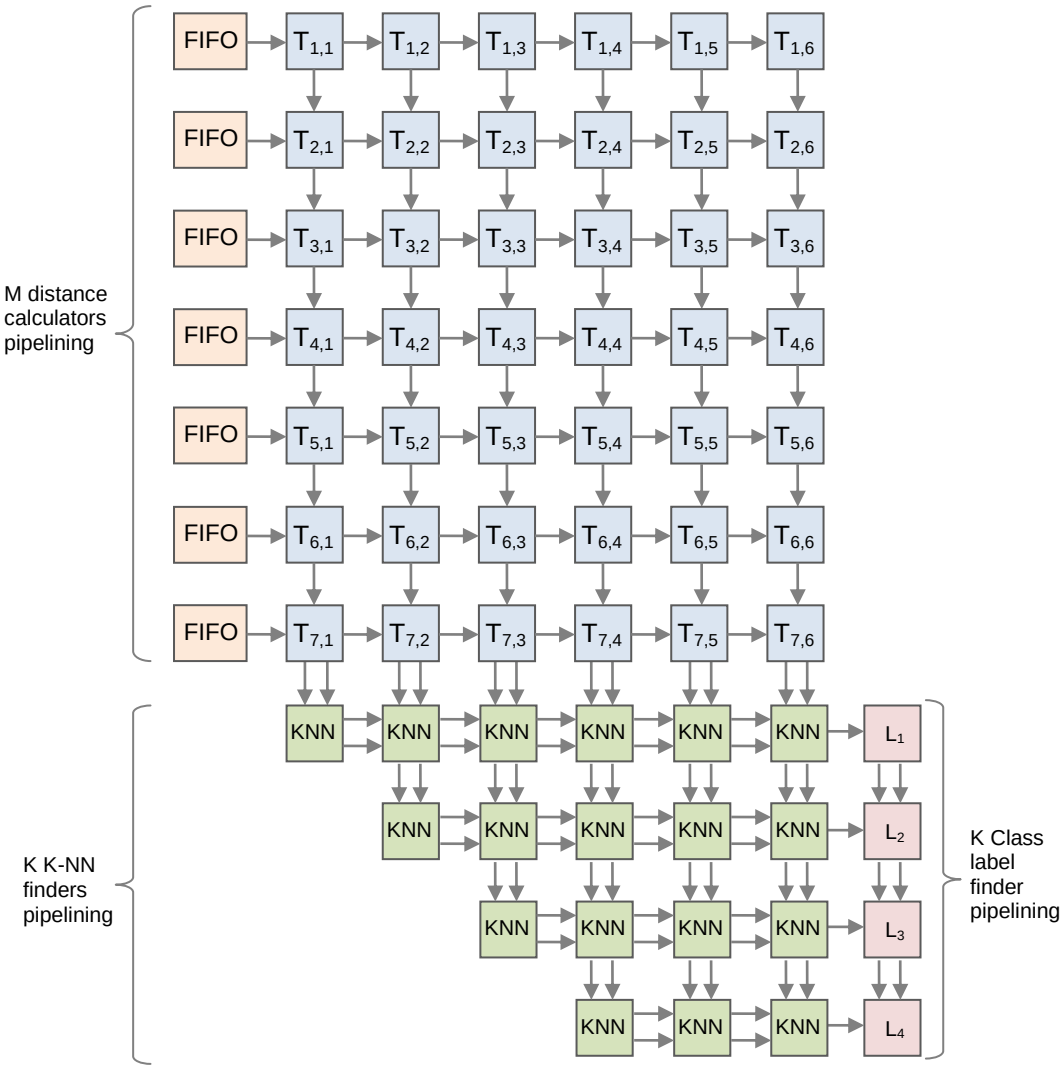


Figure 6.1 Vectors and matrixes in K-NN algorithm

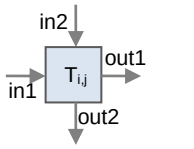
$$D(Q,T_N)=\sum_{i=1}^{i=M}\left|Q_i-T_{N,i}\right|$$ **Equation 6-1 Manhattan distance**

6.1.2 Internal Architecture of a Single K-NN Core

Our K-NN classifier mainly consists of four components: the memory block, the distance calculator, the K-NN finder, and the class label finder. The memory block is used to store the query vector and all of the training vectors. The distance calculator is responsible for computing the Manhattan distance between the query vector and all of the vectors in the training set. The distances are computed in parallel using pipelined Processing Elements (PEs), which input each sample from its attached FIFO and output the distance every clock cycle. The K-NN finder is a systolic array composed of K comparators (K-NN PEs), which populate the K minimum distances in parallel with the distance calculator. Last, but not least, the class label finder consists of a number of counters that carry on incrementing as soon as K-NNs start arriving. Figure 6.2 gives the architecture of the K-NN classifier. The classifier is composed of M distance calculators, K K-NN finders and K class label finders, where M is the number of samples in one query/training vector, and K is the number of nearest neighbours. All of the processing elements are pipelined and work in parallel (see Figure 6.2.a). In each calculation, the distance calculator subtracts the query sample from the training sample, and then adds the absolute subtraction result with the distance for the previous sample (see Figure 6.2.b). The distance for each sample is summed and the final result is populated to the K-NN finder, according to which the minimum K distances and their respective training vectors are filtered and passed to the final class label finder (see Figure 6.2.c and Figure 6.2.d). The detailed architecture schematic can be found in a previous publication [Hussain2012b].

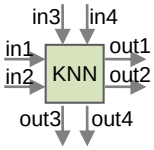


a) K-NN classifier architecture (M=7, N=6, K=4)



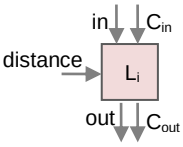
out1= in1
out2= Abs(in1-$T_{i,j}$)+in2

b) Distance calculator



out1= Min (in1,in3)
out2= in4 (In1>In3)
or out2 (In1≤In3)
out3= Max (in1,in3)
out4= in4 (In1>In3)
or in2 (In1≤In3)

c) K-NN finder



C_i= C_i +1
out= Li ($C(L_i)>C_{in}$)
or in ($C(L_i)\leq C_{in}$)

d) Class label finder

Figure 6.2 Internal architecture of a single K-NN classifier

6.1.3 Prepare a K-NN Task in R3TOS

All K-NN cores are generated from HDL code, such as in VHDL or Verilog, and then synthesised and implemented by a hardware compiler such as Xilinx Synthesis Technology (XST). In the proposed design, there are 512 training vectors and each vector consists of 6 samples; namely, N=512 and M=6. Each sample is stored in 16-bit data, and the nearest 5 neighbours are to be selected, so that K=5.

In order to be executed in the context of the R3TOS environment, certain constraints have to be applied by all K-NN cores, including wrapping with Task Control Logic (TCL), and adding area constraints.

Wrapping with Task Control Logic

The single K-NN core has to be wrapped with TCLs in order to communicate with the host and other tasks. The interface of a single K-NN core is illustrated in Figure 6.3.a.

The input data of a single K-NN core includes one query vector and 512 training vectors. Each vector is composed of 6 samples and each sample occupies 16 bits. Therefore in total (512+1) × 16 × 6 = 49248 (~50k) bits are required to store the task input data. When implemented on a Virtex-4 FPGA, all of the input data can be stored in one BRAM column. In Virtex-4 FPGAs, each BRAM column consists of 4 BRAM blocks each of which can hold up to 18k bits. Therefore there are in total 18k × 4 =72k bits of memory space available, which is enough to store the task input data. 70% of the total memory space is used by the task input, and the remaining 30% can be used as redundant space for tasks using larger parameters.

The output data of a single K-NN core includes the information about the resulting 5 nearest neighbours. In the proposed application, only the vector IDs of the 5 nearest neighbours are recorded, giving the address of the training vector in the whole training set. Because each ID is used to represent a number ranging from 0 to 511, which requires at least 9 bits, 16 bits are used to store one task ID. Each 16-bit ID can be stored in one distributed RAM. The distributed RAM is implemented on

LUTs with a different configuration. In Xilinx FPGAs, the basic element, which is the LUT, can be configured to three types of resources. The three types of resources are the basic LUT, the distributed RAM, and the shift register. The basic LUT is equivalent to a read-only memory, which is used for logic functionalities, such as a multiplexer. When configured as a basic LUT, its 16-bit content cannot be changed; hence, it is used for fixed logic. The distributed RAM gives the accessibility for both reading and writing; therefore, the 16-bit content can be used as conventional memory (RAM) to store the task result. When configured as a shift register, only the most significant and least significant bits of the 16 bits are accessible, which gives a simplified and more specific function for particular usages. In the proposed design, the LUTs are configured as distributed RAMs to store the IDs of the nearest neighbours. In total, five distributed RAMs are required to store the five neighbours.

Apart from the task data, control logic is required to synchronise tasks with the host CPU. The control signals for a single K-NN core include the start input signal, the semaphore input signal, and the finish output signal. Both of the two input signals are connected with two outputs from two LUTs; namely, the start LUT and the semaphore LUT meanwhile the output signal is connected to the input of the finish LUT. In order to avoid a situation where a memory is modified by both task logic from the functional layer and the ICAP from the configuration memory, the start input is a signal used to activate the execution of the task, and the semaphore is a signal used to maintain the coherence of distributed memory. When a task finishes its computation, it will write the finishing signal to the finish LUT, and the result will be stored in the distributed RAMs.

The input TCL contains all of the control signals and data needed for the inputs of the K-NN core, while the output TCL includes all the task's outputs (see Figure 6.3.a). The input TCL is implemented on the basic LUTs, whose content cannot be changed by the task core, while the output TCL is implemented on the distributed RAM, which can be written by user tasks. In order to give a signal to the task's input, the main CPU needs to perform a configuration operation through the ICAP port in order to change the content of the input LUT, whereby the output of these LUTs can

be changed accordingly to give different controls to the task core. Likewise, to read the output from the task core to the main CPU, a readback operation is required to deliver the content of the distributed RAMs to the main CPU. The detailed bitstream writing and reading operation is introduced in Section 6.1.4.

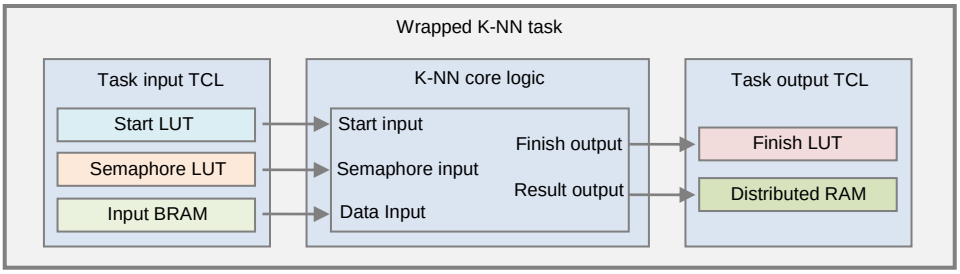
Figure 6.3 b shows the implementation of TCLs on a Virtex-4 FPGA. In Xilinx serials FPGAs, the chip is divided vertically into several clock regions, which have the same width as the chip. In each clock region, the resources are horizontally separated by resource columns (see Figure 6.3 b). Each resource column is composed of one type of resources such as IOBs, CLBs, BRAMs and DSPs. In the configuration memory, the information in one resource column is composed of a number of configuration frames, which are the smallest elements in one configuration operation. The columns for different types of resources are composed of different numbers of frames. TABLE 6.1 gives the number of blocks and frames in each type of column on a Virtex-4 and a Virtex-5 FPGA respectively. For example in Virtex-4 FPGAs, one CLB column has 16 CLB blocks, which are together composed of 22 frames. A BRAM column includes routing columns and content columns. The routing column requires 20 frames to store the connecting information of four BRAM blocks, while the content column uses 64 frames to store all the data information. In addition, one DSP column contains eight DSP blocks composed of 21 frames, and one IOB column consists of 16 IOBs which occupy 30 frames. The clock column records the configuration of the clock net in one clock region, where 3 frames are used to define the clock routing information. In the proposed application, both input and output TCLs are implemented in CLB columns. Among the 22 frames in one CLB column, the first 18 frames are used to store the routing information (see Figure 6.3 b), which is defined by the configuration of switch boxes (detailed in section 6.2). The 19^{th} and 21^{st} frames are used to store the contents of LUTs, and the 20^{th} frame records the configuration of flip-flops. In total there are 16 CLBs, and each consists of two types of slices, namely: SliceL and SliceM. Both contain two LUTs called the G-LUT and F-LUT. The difference between SliceL and SliceM is that the two LUTs in SliceL can only be configured as basic LUTs whereas the two LUTs in SliceM can be configured to any of the three types, either a basic LUT,

TABLE 6.1 BLOCK AND FRAME NUMBERS ON XILINX FPGAS

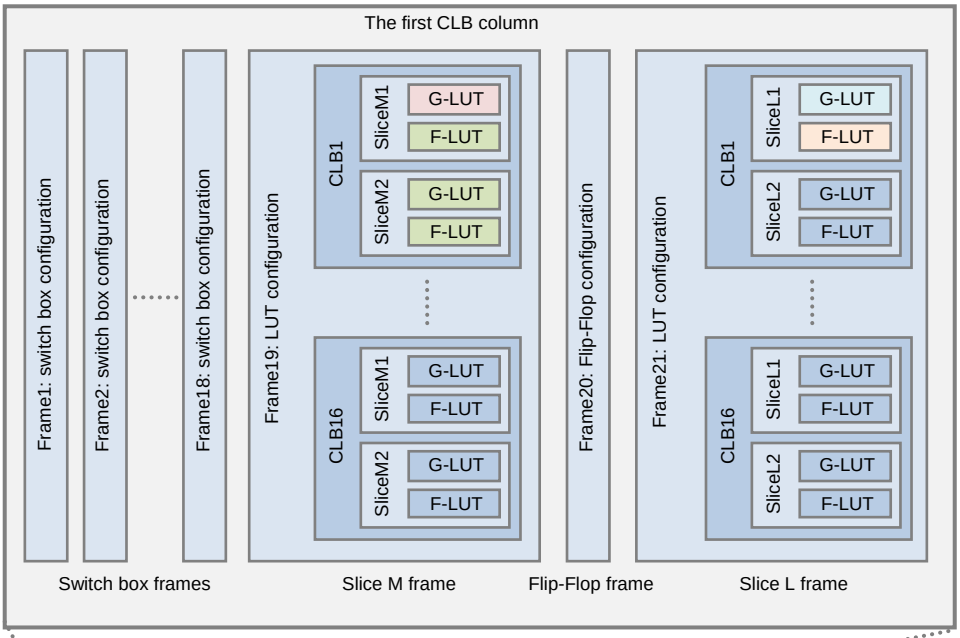
	CLB	BRAM routing	BRAM content	DSP	IOB	CLK
Virtex-4 FPGA						
Number of Blocks	16	4	4	8	16	1
Number of Frames	22	20	64	21	30	3
Virtex-5 FPGA						
Number of Blocks	20	4	4	8	20	1
Number of Frames	36	30	64	28	54	4

distributed RAM, or shift register. As a result, input TCLs are implemented in SliceL, which can only be used for circuit logic, whereas output TCLs are mapped in SliceM which is configured as distributed memory (see Figure 6.3 b). The start and semaphore LUT are mapped to G-LUT and F-LUT respectively in the first slice of the SliceL frame, while the finish LUT is mapped to G-LUT in the first slice of the SliceM frame. The output result requires 5 distributed RAMs, which occupy the 2^{nd} LUT to the 6^{th} LUT in the SliceM frame.

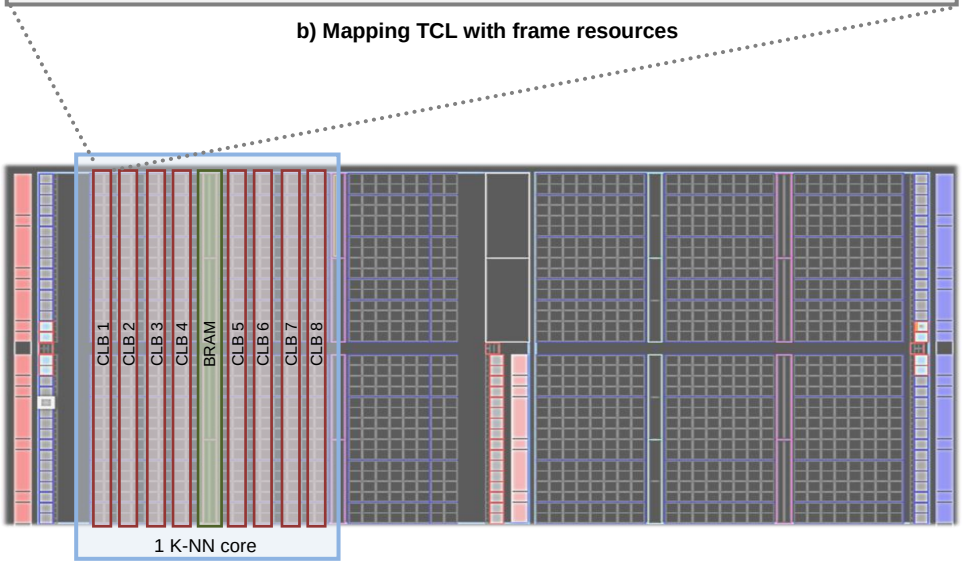
Figure 6.3 c shows the implementation of a whole K-NN core on a Virtex-4 FX12 FPGA. The implementation is constrained within one clock region, and occupies 8 CLB columns and 1 BRAM column. TCLs are allocated in the first CLB column, and the BRAM column is area-constrained in the middle of the 8 CLB columns. There are 3 BRAM columns in one clock region, and each BRAM column has at least 4 adjacent CLB columns on both its left and right sides. Therefore this area-constrained architecture is suitable for reallocation at three possible positions in one clock region.



a) K-NN task wrapped with TCL



b) Mapping TCL with frame resources



c) Implementation layout in one clock region

Figure 6.3 K-NN core wrapped with TCL and its implementation on FPGA

Adding area constraints

In order to constrain tasks as well as TCLs in the area domain, particular area constraints have to be added in the task design phase. These include an LUT position constraint, LUT connection constraint, and task area constraint.

To constrain the positions of LUTs, the individual address of each LUT needs to be specified, which can be achieved by applying various settings using synthesis tools. In the present approach, the user constraint file (ucf file) provided by the Xilinx XST tool is used. The ucf file supports several constraint syntaxes to give low-level control over the placing and routing of FPGA hardware resources during synthesis. For example, the following text constrains the start LUT in the task input TCL to be fixed at G-LUT in the slice at address X=1, Y=31:

"INST "input_TCL/start_lut_inst" BEL = G;

INST " input_TCL /start_lut_inst" LOC = SLICE_X1Y31;"

Likewise, all of the other control logic, as well as BRAMs, have to be constrained at fixed positions (see Figure 6.3), in order to facilitate bitstream manipulation in the next stage (detailed in Section 6.1.4).

The constraint on LUT connections requires low-level schematic controls. Figure 6.4 presents all the connections of task TCLs, in which the interconnections of the task input TCL and task output TCL are illustrated in Figure 6.4 a and Figure 6.4 b respectively. The task input TCL consists of two LUTs configured as basic LUTs, and two flip-flops. The two basic LUTs include one G-LUT, which is used as the task starting input at a fixed position (x1,y31), and one F-LUT that is used for the task semaphore input fixed at the same position (see Figure 6.4.a). The inputs of the two basic LUTs are connected to the ground, and the outputs are connected to two D type flip-flops, namely: FFX and FFY at position (x1, y31). The outputs from the two flip-flops are connected to the task's inputs, one for the task starting signal and the other for the task semaphore signal. The task output TCL is similar to the task input TCL, which includes two LUTs and two flip-flops (see Figure 6.4.b). However, the LUTs in the task output TCL are configured as distributed RAMs rather than basic LUTs. This allows the 16 bits in the LUTs to be changed by tasks, by

connecting the inputs of the two LUTs with the output from the task. The two distributed RAMs are fixed at position (x0, y31), which is on the left-hand side of the task input TCL. The outputs from the task are saved in the distributed RAM, waiting to be read back by the main CPU through the ICAP; and therefore the two flip-flops are disconnected from other logic.

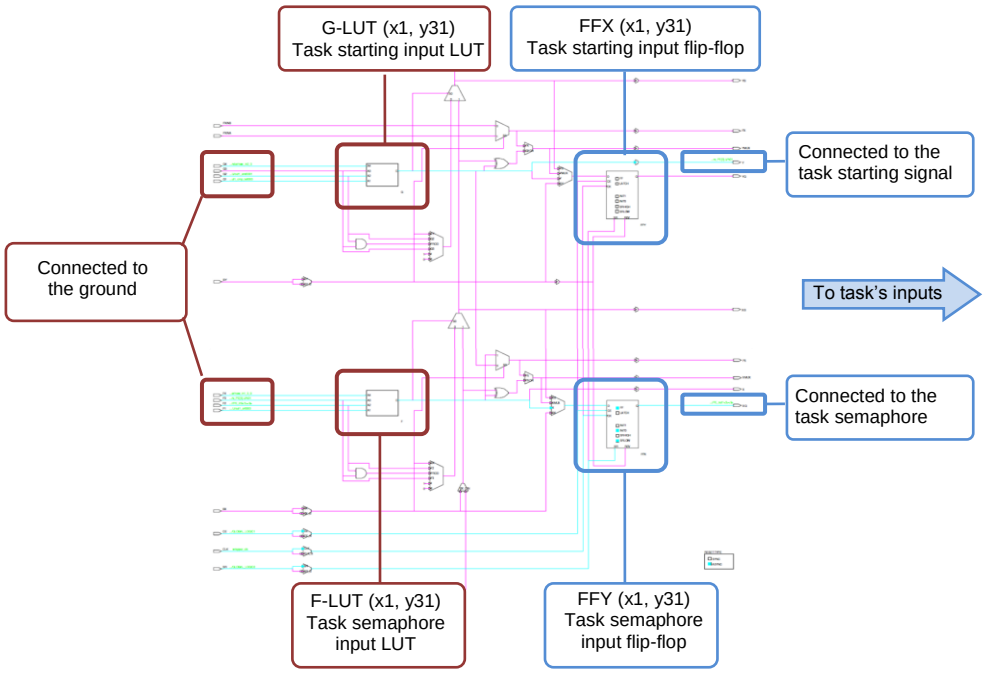
To add constraints for LUT interconnections, each LUT has to be initialised and instantiated through a structural level design. For example, the primitive of the start LUT can be instantiated by the following code:

```
start_lut_inst : LUT1
  port map (
    O => task_start_signal,    -- LUT general output
    I0 => '0'                  -- LUT input
  );
```

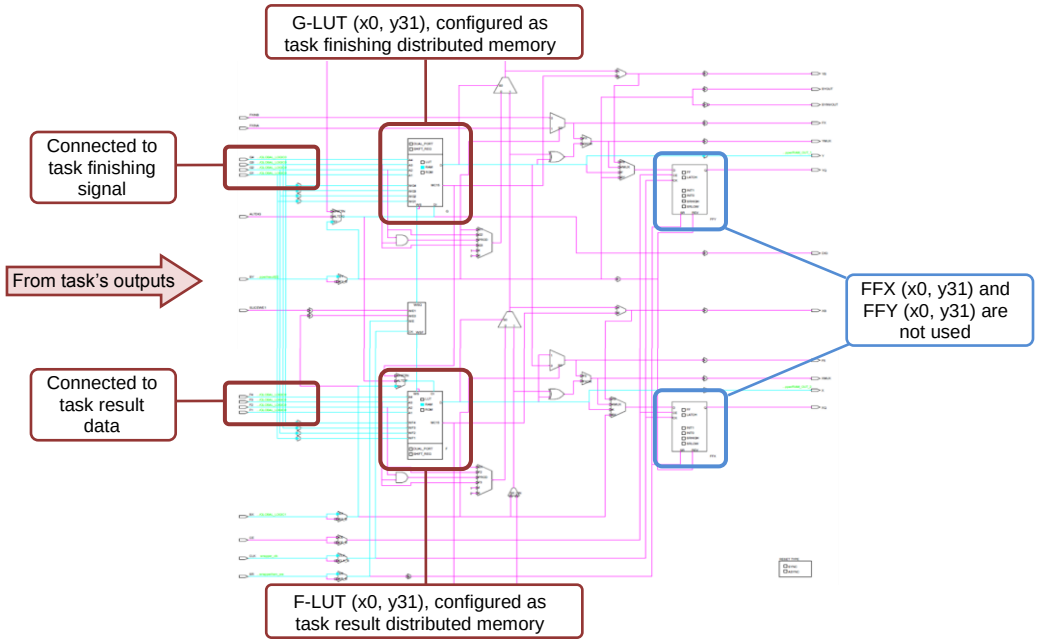
In addition, certain attributes have to be used to validate all of the LUTs used by TCLs, otherwise they will be removed from the design during synthesis optimization, since they are not treated as valid logic by the synthesiser. The reason for this is that, conventionally, if a LUT is configured as a basic LUT its output changes only according to its inputs. Therefore if all of its inputs are connected to a constant value, such as the ground, its output will remain the same all the time. In such a case, the synthesis optimiser will use a constant value to substitute this LUT in order to save resources. However, in the proposed system, the outputs of LUTs are changed by reconfiguring their content, rather than changing the inputs. Therefore, to validate LUTs instantiated by TCLs, particular attributes have to be used, such as:

```
attribute S : string;
attribute S of start_lut : signal is "true";
```

This preserves a particular LUT from being removed by the optimizer during system synthesis.



a) Connections of LUTs in task input TCL



b) Connections of LUTs in task output TCL

Figure 6.4 Constraints of LUTs' connections

6.1.4 System Operation

During the system operation time, the main CPU communicates with all K-NN tasks using the previously mentioned ICAP communication mechanism. To start a task, the main CPU firstly gives the input data to the task by reconfiguring the BRAM column through the ICAP, and then reconfigures the task starting LUT in TCL to start the task's execution. After a task finishes its computation, it saves its result into the result LUT and updates its finishing LUT. The main CPU detects the finishing signal by polling the finishing LUT once the expected execution time of the task has passed, and then reads back the resultant data from the result LUT to the main CPU memory.

When using the ICAP-based communication mechanism, all communications are achieved by modifying bitstreams through the ICAP's configuration, giving different controls to tasks by changing the underlying content in its configuration memory. However, although all of the logic functionalities are dictated by the bitstream in the configuration memory, their one-to-one mapping formats are significantly different from each other.

Figure 6.5 gives an example of mapping the logic function to its bitstream in the configuration memory. Firstly, an example of the alignment of resource columns in one FPGA clock region is presented, in which the clock region is divided by horizontally aligned resource columns spanning the same height of the clock region. The input/output blocks (IOBs) are allocated on both sides of the region in order to interface with chip externals, for example providing higher driving capabilities when outputting signals or protecting the chip from ESDs when inputting external signals. The CLB columns are the main resource distributed over most of the chip area. In addition, the clock resource crosses the whole clock region in both horizontal (HCLK) and vertical (VCLK) directions in order to configure the chip clock net. Moreover, BRAM columns are inserted among CLB columns to give more compact and denser memory blocks. As mentioned above, each CLB column is composed of 21 frames in Virtex-4 family FPGAs, which record the information about switch boxes, SliceM, flip-flops and SliceL. The 21st frame stores the information of all of

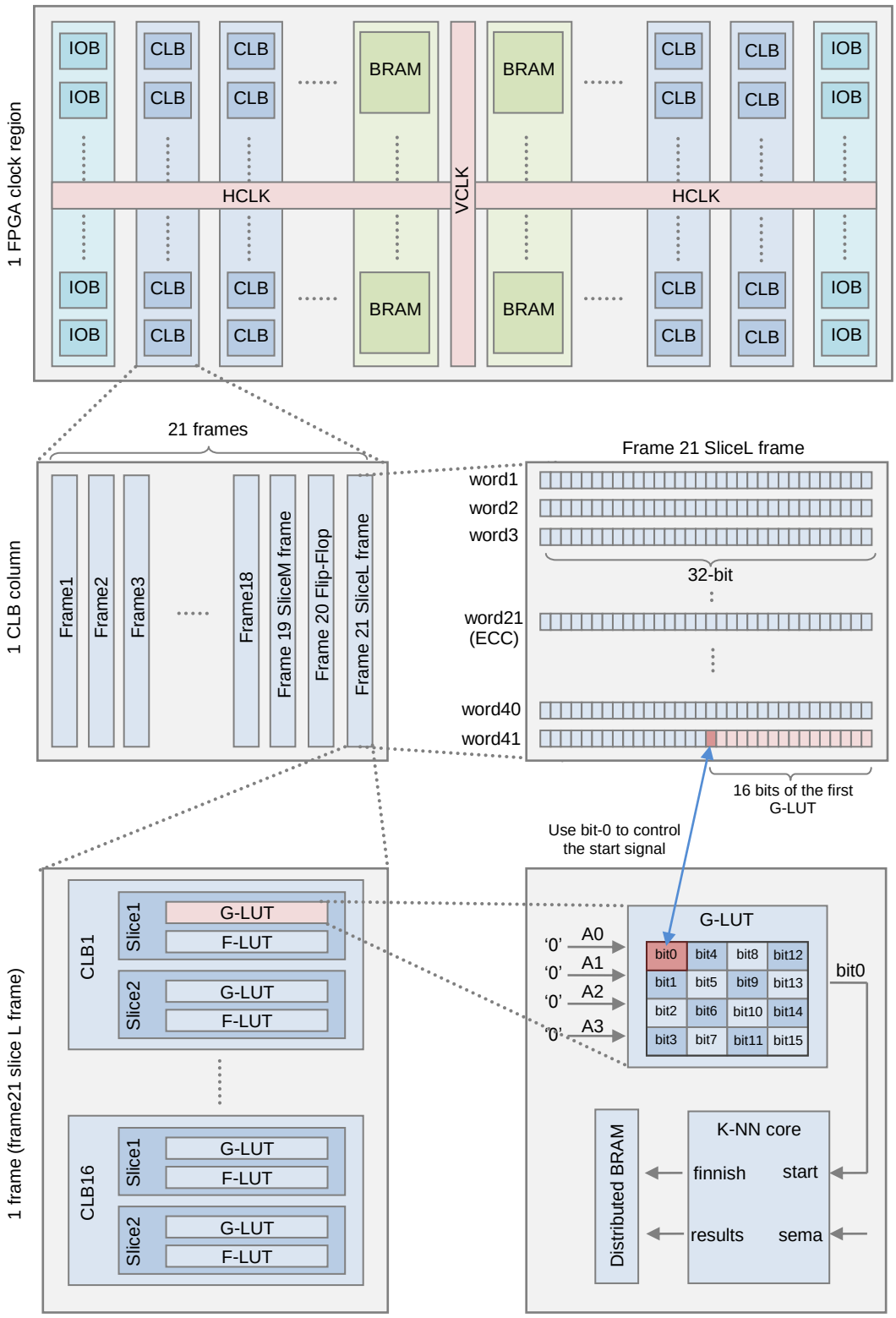
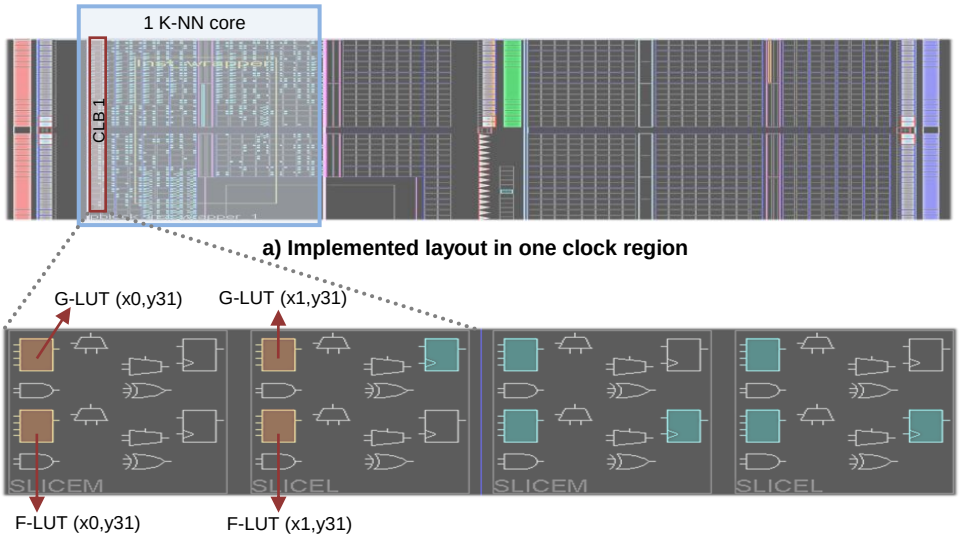


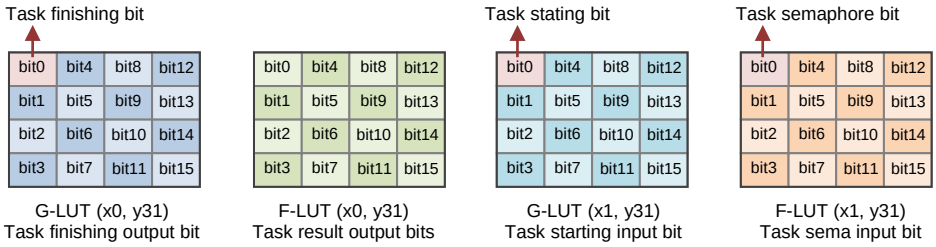
Figure 6.5 Logic functions mapped to the bitstream

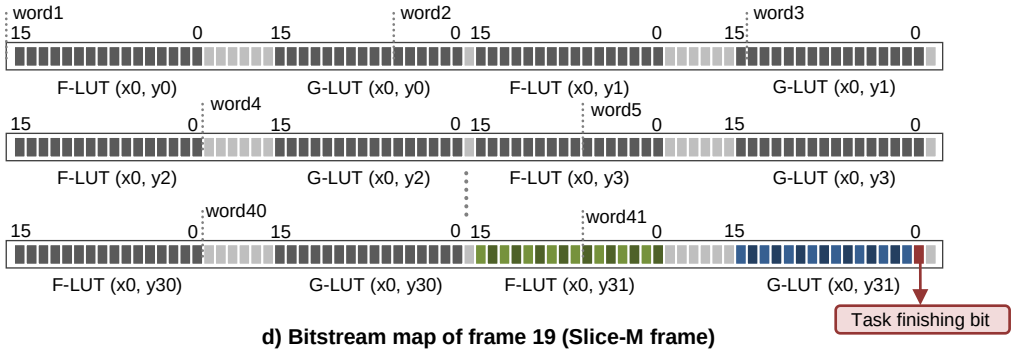
the LUTs in SliceL, which is composed of 16 CLB blocks. Each CLB block contains two slices and each slice has a G-LUT and an F-LUT. The task starting LUT is implemented in the first G-LUT of SliceL in the first CLB column. All of the inputs of this LUT are connected to the ground and its output is connected to the task starting signal. 16 bits are stored in one LUT, which are selected by the four-bit input. Since the four inputs of the starting LUT are all zeroes, the first bit will be selected to be the output, that is, output= bit0. Therefore, by changing the bit0 in the targeted LUT, the task can be started by the main CPU.

All the bits in LUTs are mapped one-to-one in the bitstream stored in the configuration memory. The configuration memory is saved in frame format. In Virtex-4 FPGAs, each frame is composed of 41 words, and each word is 32-bits wide. The contents in LUTs are distributed among all of the 41 words, while the 21st word is used to store the ECC information, which is used to check the correctness of a frame. The targeted bit0 of the first G-LUT in SliceM is allocated at the 16th bit in the 41st word. Therefore, in order to start a task, the main CPU needs to reconfigure the 21st frame in the first CLB, thus changing the 16th bit of the 41st word from zero to one. Likewise, all of the other control signals are mapped using the same principle.

Figure 6.6 gives the bitstream map of LUTs used in both input and output TCLs on Virtex-4 family FPGAs. The design layout is presented in Figure 6.6.a, which shows the implementations after placing and routing a K-NN core. The first CLB column is marked out, which includes all of the TCL logic. Figure 6.6.b is a zoom-in view of the first two CLB columns, in which the four LUTs used in TCLs are labelled, including G-LUT (x0,y31), F-LUT (x0,y31), G-LUT (x1,y31), and F-LUT (x1,y31). The 16-bit content of the four LUTs are shown in Figure 6.6.c, and the three task control bits are labelled as the task stating bit, the task finishing bit, and the semaphore bit. Note that the results from the task are stored in five LUTs, and only one of them is depicted in the figure. Figure 6.6.d and Figure 6.6.e gives the bitstream map of the 19th frame (SliceM frame) and the 21st frame (SliceL frame) respectively. In Figure 6.6.d, the 41 words of the 19th frame contain the contents of all the LUTs. The total 16 CLBs include 32 slices, which are composed of 32 G-







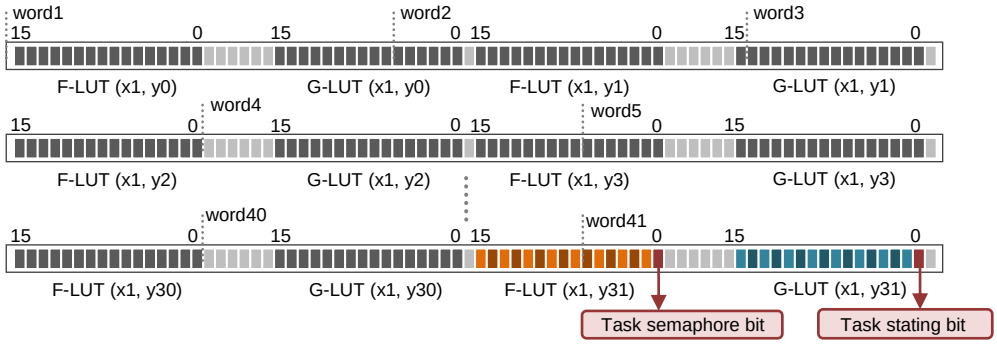
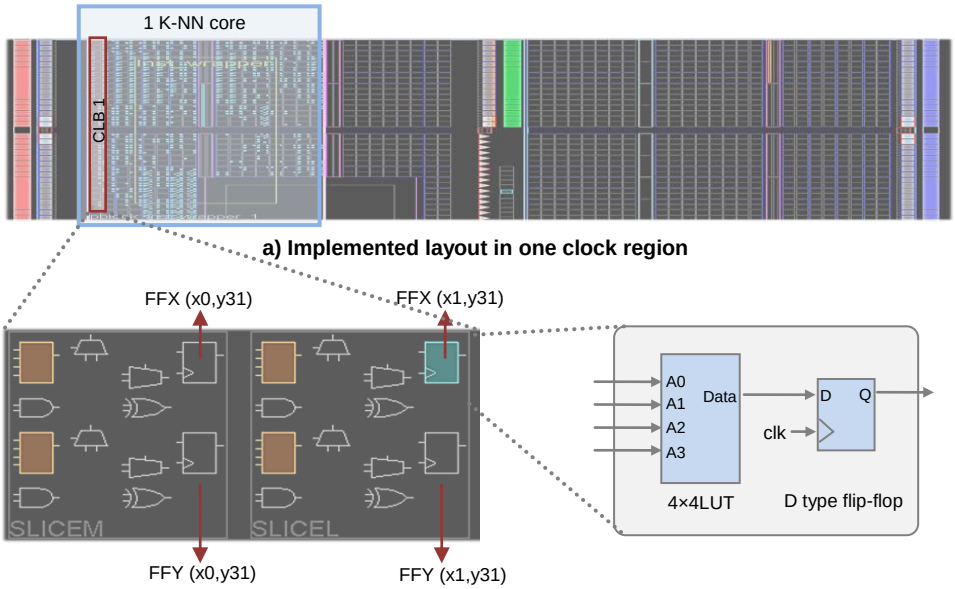


Figure 6.6 TCL LUTs mapped to the bitstream

LUTs and 32 F-LUTs, requiring in total 16×(32+32)= 1024 bits. The 41 words in the frame provide 41×32 = 1312 bits of memory to store the 1024 bits for the LUT, while the remaining bits are not used. The task finishing bit is allocated at bit1 in the 41st word, while the results are saved from the 38th word to the 41th word. Likewise, the frames in SliceL save all the LUTs' content in a similar way as SliceM. The task starting bit and the task semaphore bit are allocated at bit1 and bit22 in the 41th word respectively.

Figure 6.7 presents the bitstream map of all the flip-flops used in the input TCL. There are two flip-flops in each slice, called the FFX and FFY, and each flip-flop requires two bits to present its initial value, which is the value before starting any logic operation, and the current value which is the current state of the flip-flop. In one CLB column, there are 16 CLBs, each of which has two slices, and both SliceM and SliceL have two flip-flops. Therefore there are in total 16×2×2=64 flip-flops in one CLB column, which requires 64×2=128 bits of space in the configuration memory. All of the 128 bits are stored in the same frame, which is the 20th frame, called the flip-flop frame. This frame provides 1312 bits of space, in which 128 bits are used for flip-flops and the rest are unused. Figure 6.7 b presents the four flip-flops of the first CLB in the first column. The two flip-flops in SliceL are connected to two LUTs in the same CLB, which register the task starting bit and the task semaphore bit. Figure 6.7.c gives the frame layout of the flip-flops, which maps the state bits of all the flip-flops to their bitstream in the configuration memory. The initial value and the current state of the task semaphore bit are saved in the 41st word, while the initial and current status of the task starting bit are stored in the 40th word. The flip-flop frame gives the possibility of changing a state at run-time, which allows more dynamic control of hardware tasks, so that for example the current state of a task can be saved and restored at a later time, which is similar to context switching in the software environment. For instance, when using a preemptive scheduling algorithm, the current status of a task can be stored before the task is preempted and removed from the chip, and after other tasks with higher priorities finish their



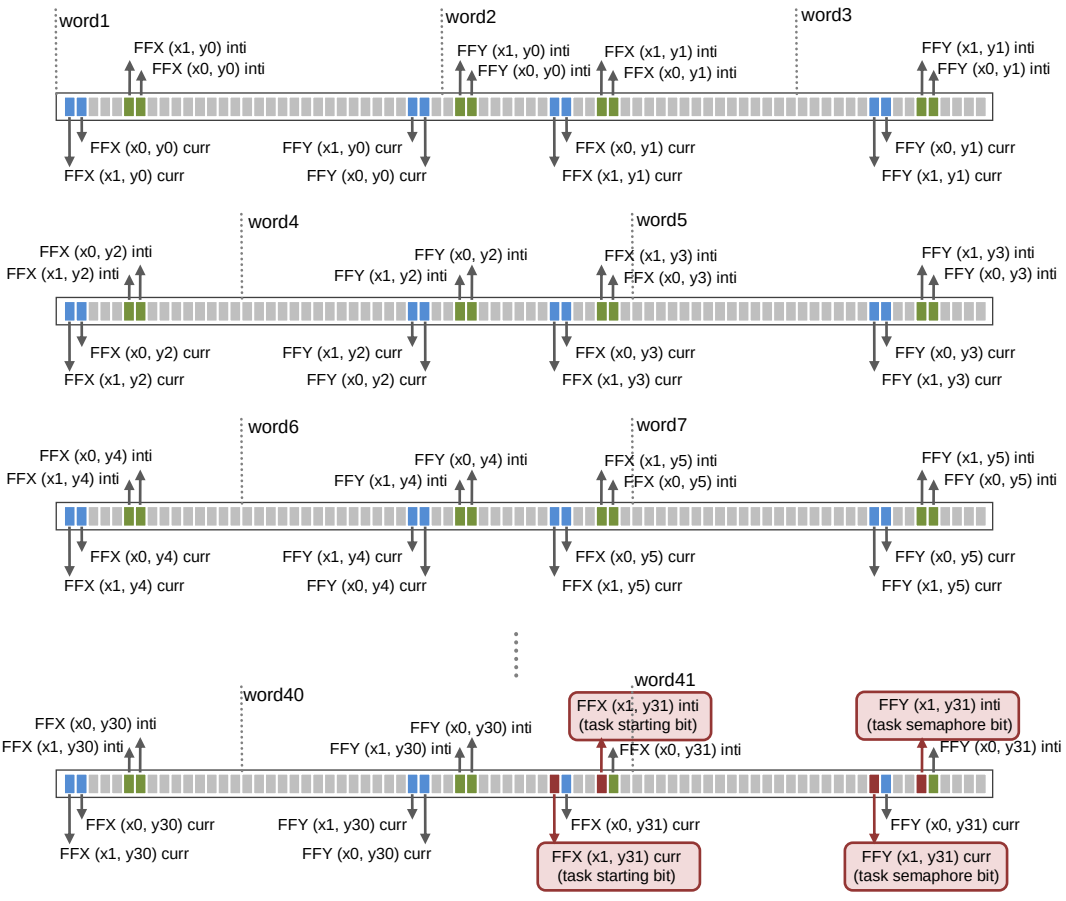
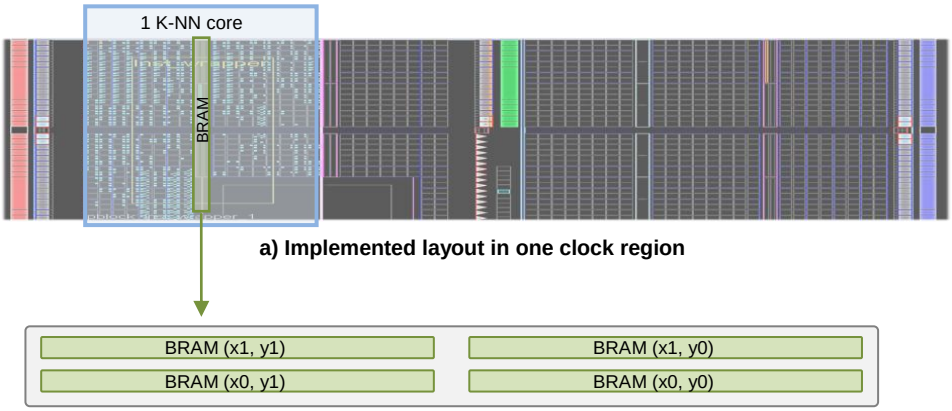


Figure 6.7 TCL flip-flops mapped to the bitstream

computation the preempted task can be resumed by reconfiguring the context back to the chip.

Figure 6.8 gives the bitstream map of the BRAM column, which is located in the middle of the K-NN core. Each BRAM column is composed of four BRAM blocks, including BRAM (x0, y0), BRAM (x0, y1), BRAM (x1, y0), and BRAM (x1, y1) (see Figure 6.8 b). The capacity of each block is 18k bits, in which data can be configured to any size varying from 1-bit to 64-bit. In Virtex-4 FPGAs, one BRAM column contains 64 frames and each frame consists of 41 32-bit words. Therefore there are 64×41×32= 83968 bits available to be used by one BRAM column in the configuration memory. On the other hand, the content of four BRAMs requires 18k×4= 72k bits, which accounts for 86% of the space available in the configuration memory. The rest of the bits are used for additional checking bits as well as control bits. In Figure 6.8 c, the three different types of bits are labelled as data bits, parity bits and saving bits. The parity bits are the checking bits used for validating the correctness of the data bits, and the saving bits are used to protect BRAM from being reconfigured. For example, before configuring the content of a particular BRAM, the saving bit of that BRAM has to be cleared in order to enable content reconfiguration. The BRAM content is saved to the bitstream in a significantly different format, so that every BRAM block occupies 10 words in all of the 64 frames (see Figure 6.8.c). The 21st word is used for the frame ECC to validate the correctness of a single frame; hence it is not used by any of the four BRAMs.



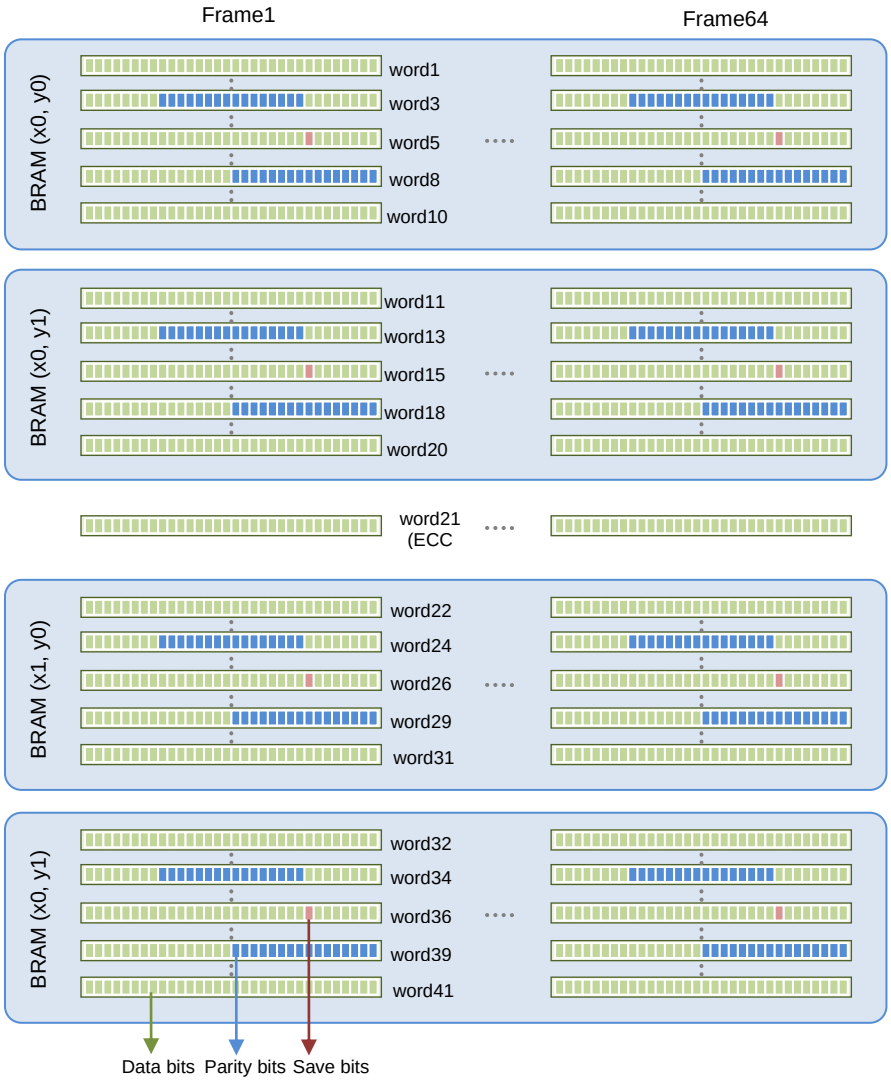


Figure 6.8 BRAM contents mapped to the bitstreams

Figure 6.9 gives the steps used to execute one K-NN task. Before allocating the task, the regional clock net has to be disabled, in order to stop all of the logic and avoid writing conflicts. Otherwise, if a BRAM is written by both on-chip logic from the functional layer and the ICAP from the configuration layer at the same time, the content will be corrupted. Therefore all the logic have to be stopped before configuring BRAMs, and this can be achieved by disabling the regional clock net of the task. Note that access to basic LUTs and other non-memory based resources do not require disabling the task. After the clock is disabled, the K-NN core will be allocated to the chip. The configuration is performed in two steps. In the first step, all of the logic functions, i.e. the CLBs are configured, and in the second step, the input data of the task is configured to the task's input BRAMs. Once both logic and inputs are configured, the regional clock will be reactivated.

In order to start a task, the task starting LUT has to be reconfigured to output a value of 1, which is the input signal to start the task. To keep the other logic unchanged while only modifying the bit in the starting LUT, the 21st frame in the first CLB column is read back before reconfiguration, whereas the other logic bits are masked out to allow this single exclusive change in the content of the starting LUT. As soon as the 21st frame is reconfigured to the chip, the task will start execution immediately.

While the task is operating, the scheduler predicts the task finishing time by checking the task's absolute finishing time. If the finishing time has passed, the main CPU will further check if the task has physically finished its computation, by polling the task's finishing bit. Note that in order to avoid a situation where the finishing bit is accessed by both the main CPU and the task's output at the same time, the semaphore bit is used to retain the coherence of the content in the distributed RAMs. To do that, the main CPU firstly reads back the 21st frame, and changes the semaphore bit (bit8 in word40) to zero. The modified frame is then written back to the configuration memory to prevent the distributed RAMs from being accessed by the task. After that, the main CPU can then safely read back the task finishing bit, as well as the results. If the finishing bit is zero, this means that the task has not finished

its computation, in which case the main CPU will release the semaphore to enable the distributed memory to be accessed by the task, and then wait until another time instance before checking the finishing bit again. If the task has finished its computation, the result will be copied from the task's output buffer (distributed RAM) to the main CPU's local memory (BRAMs). Finally, the main CPU can choose to either restart the task again using another set of input data, or remove the task from the chip by blanking of its all frames.

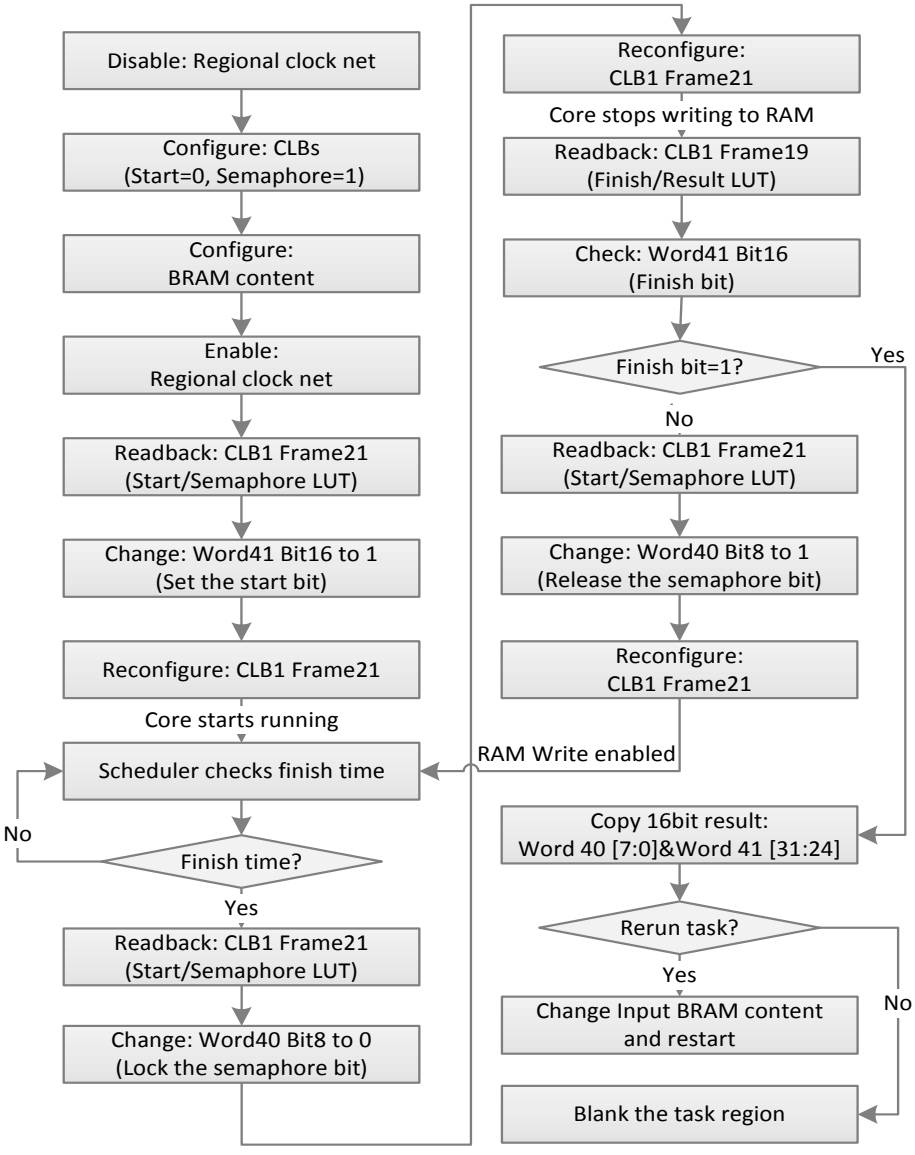


Figure 6.9 Steps to execute one K-NN task

6.1.5 Tests and Results

In the test, the system is implemented on a V4FX60 FPGA [Xilinx2008]. Figure 6.10 shows the schematic layout of the whole chip, which was generated by the Xilinx FPGA editor [Xilinx1999]. The chip consists of 8 clock regions, and each clock region has 52 CLB columns, 8 BRAM columns, 2 DSP columns, and 2 IOB columns. Figure 6.10 depicts the 6 BRAM columns that could be used to place the task, since they have 4 adjacent CLB columns on both sides. The system, including the R3TOS HWμK and the main CPU, is allocated on the top-left corner of the chip, leaving the rest of the chip for the placing of hardware tasks. An example of a K-NN task is placed at position (x44, y0), which is the 44^{th} CLB column in the first clock region. Apart from this position, there are in total 29 positions where it is possible to place another task, or to reallocate the current task. The right-hand 24 positions can be used for three FCCRs. Each FCCR is composed of 8 possible positions in the same column. If high reliability is demanded, a K-NN task can have three copies of instances separately placed in three FCCRs, giving triple redundancy for each task computation. Note that the task already placed is clocked by the regional horizontal clock line, which is already activated by the task and the system kernels. However for the other task regions, the horizontal regional clock line has to be activated before starting task execution.

Parameters of a K-NN Task

The hardware features and software features are extracted during the task design phase. TABLE 6.2 gives the features of an example of a K-NN task being executed by the R3TOS. The hardware features are used by the R3TOS HWμK, including its time and area information, while the software features allow the task to be treated as a traditional software task in the R3TOS's main CPU. The task ID indicates that the task is the first task in the task queue, and the next task ID points to the next task in the same queue. The task is currently in executing status and the task deadline is pre-set to 500 μs, which gives relatively high redundancy. The task arrives as soon as the system starts operation, and then it is allocated to the chip, which requires 98 μs. The predicted task finishing time is the sum of the absolute allocated time and the task's

TABLE 6.2 HARDWARE AND SOFTWARE FEATURES OF A K-NN TASK

Hardware Features		Software Features	
Task_ID	*1*	*Function_Name*	*Compute_KNN*
Next_Task_ID	*2*	*Input_Arguments*	*K= 5, M= 6, N= 512*
Task_Status: Executing	*Executing*	*Input_Memory_Address*	*0x0200*
Absolute_Deadline	*500 µs*	*Output_Result*	*KNN_Res (5 bytes)*
Task_Arriving_Time	*0*	*Output_Memory_Address*	*0x0300*
Task_Allocating_Time	*98 µs*	*Task_Status*	*Executing*
Task_Execution_Time	*37 µs*		
Task_Finishing_Time	*135 µs*		
Task_Width	*9*		
Task_Height	*1*		
Task_Resource_Type	*000010000*		
Task_Area	*9×1*		
Task_Allocated_Position	*(x44,y1)*		

executing time. Note that the K-NN task has a deterministic execution time of 37 µs, clocked by a 100MHz system clock. In the area domain, this K-NN task requires 8 CLB columns and 1 BRAM column, which span the height of a clock region; therefore the width, the height, and the size of the task are 9, 1, and 9×1 respectively. The task has been allocated at position (x44, y1), occupying the 36^{th} CLB to the 44^{th} CLB columns. From a software perspective, the task is defined as a traditional software task, which has a function name, input arguments, and returned results. The returned result of the task is a user-defined structure called *KNN-Res*, which consists of 5 bytes to present the 5 nearest neighbours. In addition, the task has two pointers to the addresses of the input data and the output data, which are located at addresses *0x0200* and *0x0300* respectively. Moreover, the current status of the task is also available to be checked by the main CPU from the software environment.

Reliability Test

To test the reliability of the system, random faulty bits are injected into the system's operation. The faults are injected at a rate of 1bit/s rate using the ICAP manager. To detect the injected faults, the scrubbing technique is applied to read back the full bitstream from the configuration memory. The whole chip consists of 25,280 slices, which requires 550,226 cycles (~5 ms at 100MHz system clock) for full-chip

scrubbing. The scrubbing is operated at a frequency of 100Hz, which is 100 times full-chip scrubbing every second, and this is sufficient to cater for the possibility of emerging damage and SEUs [Fuller2000]. In the test, a K-NN task is triplicated to three copies of instances, which are initially placed in three FCCRs at positions 3, 4, and 5. During system operation time, three instances are running in parallel at the same time, and faults are periodically injected at random positions on the chip. The results show that a fault can be detected within less than 10ms. If a fault is injected into one of the task instances, the majority voter can detect the error and reconfigure the faulty instance in the specific FCCR. A damaged resource can be emulated by repeatedly injecting faults to the same bit more than twice. In such a case, the task will be reallocated and restarted at a new position, which is calculated by the task allocator using the EAC algorithm.

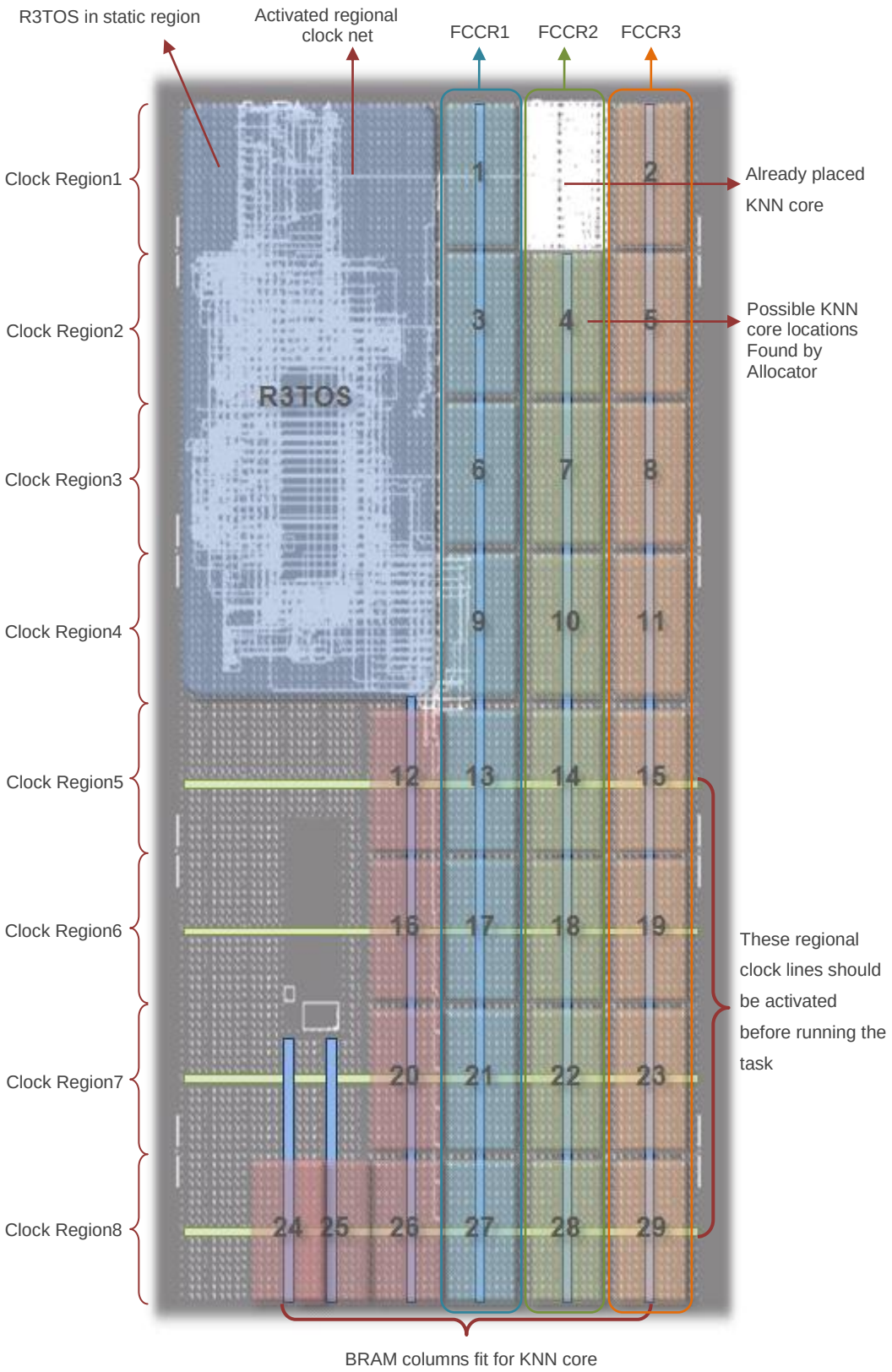


Figure 6.10 R3TOS and task placement on a V4FX60 FPGA

Performance Analysis

TABLE 6.3 presents the evaluation of the performance of the task allocator, giving measurements of the computation time of the EAC algorithm when implemented on real hardware coded in the PicoBlaze microprocessor. The execution time includes the time to find the best location, and the time to update matrices when inserting or removing a task. In addition, to increase the speed of the algorithm, the EAC First-Fit (FF) algorithm is implemented, according to which the task is allocated at the first possible position instead of the best position, whereby computation time is reduced at the cost of chip compactness. On average, for an 8×8 chip granularity, the allocator requires only 41 μs to find the best location when running at 100MHz clock frequency. After the position is decided, the allocator needs 232 μs to update matrices when inserting a task, or 210 μs to remove a task. When using the FF algorithm, the time to find a position is reduced to 3.5 μs, while the updating time remains the same as with the Best-Fit EAC. The algorithm's computation time increases with chip granularity, so for example the computation time for 10×10 granularity is about twice that of 8×8 granularity. In the proposed application, although there are 8×52 CLB columns, the actually possible positions can be modelled as an 8×6 matrix, so that the algorithm requires less computation.

The performance of the ICAP manager is presented in TABLE 6.4, in which the time and the number of operating cycles are given for different types of operations. The allocation of a single K-NN task requires the configuration of 8 CLB columns, which needs 6920 clock cycles equivalent to 69.2 μs when running at 100MHz system frequency. To give the data to the task's input BRAM, a full BRAM column needs to be reconfigured, which takes 2656 cycles or 26.6 μs at 100MHz clock frequency. After both CLBs and BRAMs are configured, the task can be started by reconfiguring the input TCL, which includes the starting bit and the semaphore bit. Since the writing operation requires two steps, reading back the bitstream and masking out irrelevant bits, before reconfiguring the target frame, the operation requires 238 cycles or 2.38 μs at 100MHz clock frequency. After the task finishes its computation, the result can be read back by the ICAP manager, which needs 114

clock cycles or 1.14 μs at 100MHz clock frequency. A finished task can be removed by blanking the task region; namely, writing all-zero frames to the task. The blanking frames can be configured using Multiple Frame Writing (MFW), whereby the speed is increased and the operation can be completed within 704 cycles or 7.04 μs at 100MHz clock frequency. Likewise, MFW can also be applied to replicate multiple K-NN cores to increase configuration speed, where for example replicating 10 K-NN cores requires 13640 cycles or 136.4 μs at 100MHz clock frequency.

The configuration speed of the system is compared with that of the Xilinx IP core; namely the *HWICAP* [Xilinx2010b]. To configure or readback a single frame, the *HWICAP* requires more than 300 μs due to its setup overhead in both hardware and software, whereas the system proposed here can configure/readback one single frame in ~1 μs. Therefore the configuration speed of R3TOS has achieved a speed-up of ~300x compared with *HWICAP* driver [Ebrahim2012].

On the other hand, benefiting from the high speed performance, the power consumption has also been significantly reduced. The power consumption of the K-NN hardware task has been compared with its implementation on a General Purpose Processor (GPP), and results show that a single core consumes about six times less power than an equivalent implementation on GPPs. The detailed results have been published by group member Hanaa M. Hussain in her doctored thesis [Hussain2012b]

TABLE 6.3 ALLOCATOR EXECUTION TIME BREAK-DOWN

Chip granularity	8×8	10×10	16×12
Finding Best Location Average Time	41 μs	118 μs	465 μs
Corresponding Operations	3.85 k	9 k	28.15 k
Insert Task Updating Average Time	232 μs	674 μs	2191 μs
Corresponding Operations	11.6 k	33.7 k	109.55 k
Remove Task Updating Average Time	210 μs	630 μs	1866 μs
Corresponding Operations	10.5 k	31.5 k	93.3 k
Finding First Location Average Time (FF)	3.5 μs	4.25 μs	9.8 μs
Corresponding Operations	0.18k	0.21k	0.49k

TABLE 6.4 ICAP MANAGER EXECUTION TIME BREAK-DOWN

Process	Cycles	Time
Allocate 1 K-NN Task (CLBs)	6920	69.20 μs
Configure 1 BRAM Column	2656	26.6 μs
Write to input TCL (Start/Semaphore)	238	2.38 μs
Read from output TCL (Finish/Result)	114	1.14 μs
Blanking 1 K-NN Core	704	7.04 μs
Replicate 10 K-NN Cores	13640	136.40 μs

Area Consumption

The resource consumption of a single K-NN task , including the K-NN core and TCL logic presented in TABLE 6.5. One K-NN task (K= 5, M= 6, N= 512) consumes in total 650 slices, which includes 1060 4-input LUTs and 708 flip-flops. In addition, 4 CLB blocks are used to store the task's input data.

TABLE 6.6 gives the resource break-down of the R3TOS system kernels. The ICAP manager includes the PicoBlaze kernel, the hardware ICAP driver and the bitstream BRAM, which together occupy 457 slices and 2 BRAMs (one for the bitstream and the other for the PicoBlaze program memory). The task allocator is based on a single PicoBlaze core which requires 96 slices and 2 BRAMs (one for storing matrices and the other for the PicoBlaze program memory). The task scheduler contains additional logic to communicate with the allocator and the ICAP manager; and therefore it occupies 138 slices and 2 BRAMs (one for storing tasks and the other for the PicoBlaze program memory). The main CPU is implemented on a MicroBlaze microprocessor, which requires 924 slices for its logic and 8 BRAMs for its data and instruction memory. Put together, the whole R3TOS system requires 1615 slices and 14 BRAM columns, which accounts for only 10% and 6% respectively of the total number of slices and BRAMs on a Virtex4 FX60 FPGA.

TABLE 6.5 RESOURCE CONSUMPTION OF A SINGLE K-NN TASK

Resources	Number of occupied resources
Slices	650
4 input LUTs	1060
Slice Flip Flops	708
BRAMs	4

TABLE 6.6 R3TOS SYSTEM RESOURCE BREAK-DOWN

Component	Slices	BRAMs
ICAP manager	457	2
Task Allocator	96	2
Task Scheduler	138	2
Main CPU	924	8
Total R3TOS	1615	14
Percentage of Virtex4 FX60	10%	6%

6.2 Case Study 2 – Sequence Alignment (SA)

In bioinformatics, sequence alignment is used to differentiate between two biological sequences so as to identify the similarities between two DNA, RNA, or protein sequences [Gotoh1982, Durbin1998]. The analysis of a gene sequence always requires heavy computation. As a result, traditionally used single core processors, such as CPUs, consume large amount of time. In contrast, multicore or parallel computing based platforms such as GPUs and FPGAs achieve better computational efficiency and therefore are nowadays widely used in sequence alignment applications [Oliver2005, Benkrid2012].

On the other hand, in some server-based, multi-user, multi-tasking applications, computing resources can be shared by different users and different application tasks at the same time, so that better resource usage efficiency can be achieved [Hong2013a, Hong2013b]. Aiming to achieve this, the R3TOS is enhanced to be able to customise tasks at run-time, which is demonstrated in the context of the sequence alignment application [Hong2013b]. Here, the size of tasks can be dynamically adjusted in response to the currently availability of resources, whereby the hardware resources can be more efficiently used. This application demonstrates an additional functionality of the R3TOS of on-line task assembly, which allows tasks to be generated from single Processing Elements (PEs) at run-time in order to improve the efficiency of resource usage.

The following sections firstly give a review of sequence alignment algorithms, and then illustrate the folding technique and the implementation of the proposed reconfigurable PE architecture. After that, the PEs are integrated with the system, which requires particular design constraints to enable the PEs to be assembled at run-time. In addition, the multiple frame writing (MFW) technique is used to increase the configuration speed. Finally, tests and results are presented to evaluate the system performance.

6.2.1 Pairwise Biological Sequence Alignment

Pairwise biological sequence alignment has been widely used in bioinformatics applications, requiring the regional/global similarities between two biological sequences to be identified [Isa2012]. To find the optimum alignment, dynamic programming-based sequence alignment algorithms have been widely applied. One typical such algorithm is the Smith-Waterman (S-W) algorithm, which uses a matrix *F(i, j)* to find and score the best alignment between a query sequence and database sequences [Durbin1998]. The matrix is calculated recursively using:

$$F(i,j) = \max \begin{cases} 0 \\ F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases}$$

Equation 6-2 Score using h lineal gap penalty

In this equation, $S(x_i,y_j)$ is the probabilistic score, which describes the biological relationship of amino acids between x_i and y_j. The probabilistic score can be obtained by substituting x_i with y_i or vice-versa, from the *BLOSUM50* substitution matrix (see Figure 6.11) or other amino-acid probabilistic models such as *BLOSUM62* and *PAM* [Durbin1998]. The linear gap penalty *(d)* is a constant value which gives linear penalties for the insertion or deletion of residues, where *penalty (g)* = $- g \times d$.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	**5**	−2	−1	−2	−1	−1	−1	0	−2	−1	−2	−1	−1	−3	−1	1	0	−3	−2	0
R	−2	**7**	−1	−2	−4	1	0	−3	0	−4	−3	3	−2	−3	−3	−1	−1	−3	−1	−3
N	−1	−1	**7**	2	−2	0	0	0	1	−3	−4	0	−2	−4	−2	1	0	−4	−2	−3
D	−2	−2	2	**8**	−4	0	2	−1	−1	−4	−4	−1	−4	−5	−1	0	−1	−5	−3	−4
C	−1	−4	−2	−4	**13**	−3	−3	−3	−3	−2	−2	−3	−2	−2	−4	−1	−1	−5	−3	−1
Q	−1	1	0	0	−3	**7**	2	−2	1	−3	−2	2	0	−4	−1	0	−1	−1	−1	−3
E	−1	0	0	2	−3	2	**6**	−3	0	−4	−3	1	−2	−3	−1	−1	−1	−3	−2	−3
G	0	−3	0	−1	−3	−2	−3	**8**	−2	−4	−4	−2	−3	−4	−2	0	−2	−3	−3	−4
H	−2	0	1	−1	−3	1	0	−2	**10**	−4	−3	0	−1	−1	−2	−1	−2	−3	2	−4
I	−1	−4	−3	−4	−2	−3	−4	−4	−4	**5**	2	−3	2	0	−3	−3	−1	−3	−1	4
L	−2	−3	−4	−4	−2	−2	−3	−4	−3	2	**5**	−3	3	1	−4	−3	−1	−2	−1	1
K	−1	3	0	−1	−3	2	1	−2	0	−3	−3	**6**	−2	−4	−1	0	−1	−3	−2	−3
M	−1	−2	−2	−4	−2	0	−2	−3	−1	2	3	−2	**7**	0	−3	−2	−1	−1	0	1
F	−3	−3	−4	−5	−2	−4	−3	−4	−1	0	1	−4	0	**8**	−4	−3	−2	1	4	−1
P	−1	−3	−2	−1	−4	−1	−1	−2	−2	−3	−4	−1	−3	−4	**10**	−1	−1	−4	−3	−3
S	1	−1	1	0	−1	0	−1	0	−1	−3	−3	0	−2	−3	−1	**5**	2	−4	−2	−2
T	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	2	**5**	−3	−2	0
W	−3	−3	−4	−5	−5	−1	−3	−3	−3	−3	−2	−3	−1	1	−4	−4	−3	**15**	2	−3
Y	−2	−1	−2	−3	−3	−1	−2	−3	2	−1	−1	−2	0	4	−3	−2	−2	2	**8**	−1
V	0	−3	−3	−4	−1	−3	−3	−4	−4	4	1	−3	1	−1	−3	−2	0	−3	−1	**5**

Figure 6.11 BLOSUM50 substitution matrix

Figure 6.12 gives an example of calculating the score matrix using the S-W algorithm, in which a query sequence N is compared with database sequence M, using a linear gap penalty of *d*=*3*. After all of the scores are populated, the final score reflects the similarity of the two sequences, and the optimum alignment can be retrieved by tracking back the populating route numbers marked in bold.

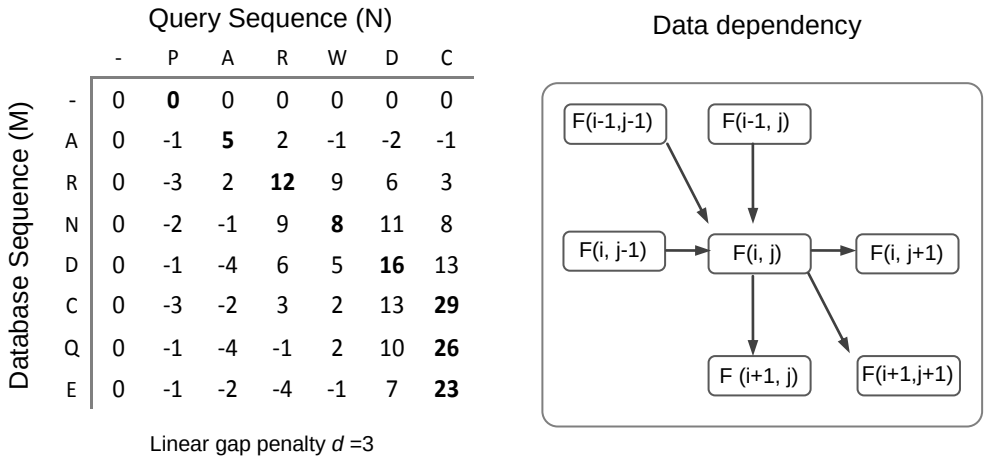


Figure 6.12 Smith-Waterman algorithm with lineal gap penalty

The affine gap penalty was introduced by GOTOH in 1982, and its use can obtain more biologically realistic results compared with the linear gap penalty [Osamu1982]. The affine gap penalty extends the score to three matrices:

$$F(i,j)=\max\begin{cases}F(i-1,j-1)+s(x_i,y_j)\\ I_x(i-1,j-1)+s(x_i,y_j)\\ I_y(i-1,j-1)+s(x_i,y_j)\end{cases}$$

$$I_x(i,j)=\max\begin{cases}F(i,j-1)-d\\ I_x(i,j-1)-e\end{cases}$$

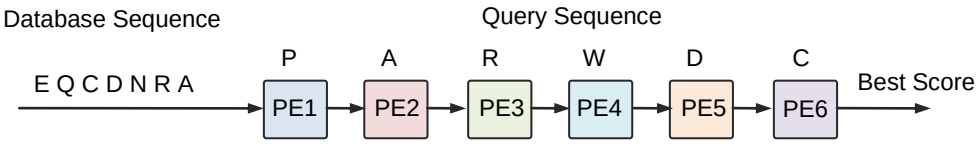
Equation 6-3 Score using affine gap penalty

$$I_y(i,j)=\max\begin{cases}F(i-1,j)-d\\ I_y(i-1,j)-e\end{cases}$$

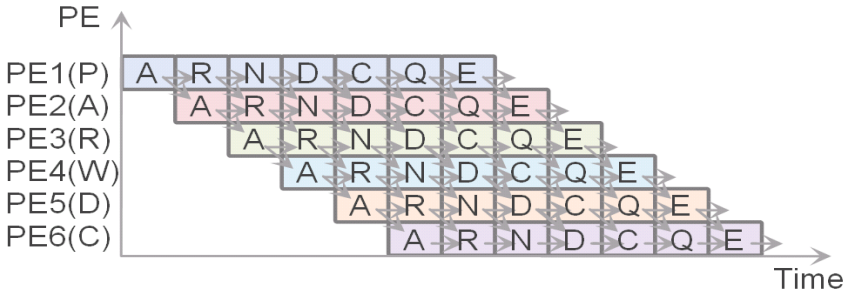
When using the affine gap penalty, *F(i,j)* refers to the best score so far, and $I_x(i,j)$, and $I_y(i,j)$ are the scores aligned to gaps with residues x_i and y_i respectively. The affine gap penalty penalises gaps depending on both gap number and gap length, so that *penalty (g) = - d- (g- 1)* × *e*, where *d* is the penalty for opening a new gap and *e* is the penalty associated with extending a gap. In the proposed application, the affine gap penalty is implemented since it gives more biologically realistic results [Isa2012].

6.2.2 Hardware Pipeline and PE Folding

The S-W algorithm can be implemented in pipelined PE arrays using on-chip hardware logic. All of the residues in the query sequence of length N are mapped one-to-one to the PE array of length N, while the database of length M is shifted into the array on every clock cycle. The highest score can be populated after the entire database sequence is shifted through the PE array. Figure 6.13 a gives pipelined PE arrays to align the two sequences shown in Figure 6.12. The query sequence consists of 6 residues (‘P’, ‘A’, ‘R’, ‘W’, ‘D’, and ‘C’), which are mapped with 6 PEs (PE1 to PE6 respectively). The database sequence is composed of 7 residues (‘A’, ‘R’, ‘N’, ‘D’, ‘C’, ‘Q’, and ‘E’), which is shifted into the PE array. The best score is obtained



a) Pipelined PE array



b) Processing datapath

Figure 6.13 Pipelined PE array without using PE folding

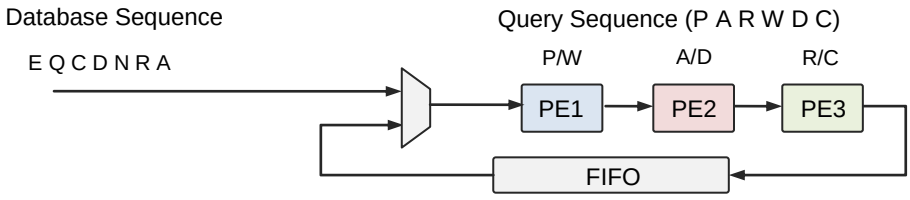
from the last PE after the whole database sequence is shifted through. Figure 6.13.b gives the datapath of the PE array, in which each PE inputs the previous results from both the previous PE and itself, and outputs the current result to the next PE. The computation starts as soon as the first residue is shifted in, and finishes when the last residue is shifted out, which requires 14 computing stages. For a query sequence of length N and a database sequence of length M, the pipelined hardware requires (M+N+1) stages to align an M×N sequence, which is one order of time faster than software sequential programming which needs M×N steps.

PE Folding Mechanism

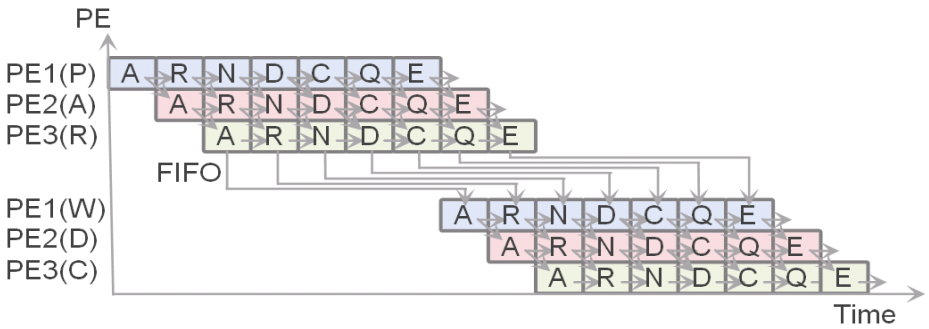
Since the hardware resources are often in practice limited, it is difficult to allow for one-to-one mapping between PEs and the query residues. Therefore, in order to compute larger query sequences, limited resources have to be reused by the PE folding technique.

Figure 6.14 a gives an example of PE folding, in which the previous sequences are aligned by using only three PEs. In the first iteration, the three PEs are assigned with

the first three residues in the query sequence, ‘P’, ‘A’, and ‘R’. After the entire database sequence is shifted in, the results are buffered into a FIFO, and the three PEs are then folded with the second set of residues in the query sequence, ‘W’, ‘D’, and ‘C’. In the second iteration, the inputs of the three PEs are multiplexed from the FIFO, which buffers the results of the first iteration. After all of the residues in the query sequence have been folded, the final results will be obtained from the last PE. Figure 6.14.b shows the datapath of the PE array when using PE folding. As soon as the first PE (PE1) finishes its first iteration, it starts receiving data from the last PE (PE3). Since every PE needs to shift the entire database sequence twice, the total alignment takes 18 stages to complete. For a query sequence of length N, a database sequence of length M, and the folding number of K, the PE array requires (K×M+N/K+1) stages to align M×N sequence. By using the folding technique, more resources can be saved at the expense of performance in terms of the speed of computation.



a) Folded PE array



b) Datapath of folded PE array

Figure 6.14 Pipelined PE array using PE folding

Single PE Architecture

In order to allow for task on-line generation as well as PE folding, the architecture has been designed for a single reconfigurable PE [Isa2012, Isa2013]. Figure 6.15 gives the internal architecture for a single PE. There are four inputs and four outputs for each PE. The four inputs include $F(i,j-1)$, $I_x(i,j-1)$, $M(i,j-1)$, and x_i, which respectively represent: the best score from the previous PE; the score aligned to gaps with residue x_i from the previous PE; the best score updated by the previous PE; and the database sequence. The four outputs are $F(i,j)$, $I_x(i,j)$, $M(i,j)$, x_i, which respectively stand for: the best score computed by the current PE; the score aligned to gaps with residue x_i from the current PE; the best score updated by the current PE; and the database sequence shifted to the next PE respectively. The internal signals include the values calculated by the same PE in the previous iteration ($F(i-1, j-1)$, $I_x(i-1, j-1)$, $I_y(i-1, j-1)$, $I_y(i-1, j)$, and $F(i-1, j)$) , and the gap penalty constants GPE and GPD which are negative values to penalise the opening of a gap and extending an existing gap respectively. All the scores are registered in 11 bits registers, and the gap penalties are stored in 4-bit data. Since all the elements are internally pipelined in a single PE, the output results can be populated on every clock cycle, after the 3-cycle latency at the beginning of its execution. The detailed schematic level implementation has been published elsewhere [Isa2013].

Each PE uses two Configuration Elements (CEs) to store the substitution values, one used for the current computation while the other can be concurrently reconfigured to the query residue to be used in the next fold. Each CE is implemented in a number of LUTs in which the similarity table, which contains the substitution values for a particular residue, is stored (see Figure 6.11). Figure 6.16 gives an example of the two CEs used for PE folding. In this example, the PE needs to be folded three times, to map residues ‘A’, ‘D’, ‘C’ respectively. In the first folding, the PE is mapped with residue ‘A’, whose substitution values are stored in CE0. While CE0 is used for computing, CE1 is being configured to residue ‘D’, which will be used for the second fold. Likewise, while CE1 is used for computing residue ‘D’ in the second fold, CE0 is being configured to the third residue ‘C’ at the same time. Each CE

inputs a 5-bit residue value from the query sequence, and uses it to look up the substitution value stored in the similarity table, and then outputs the substitution value to the multiplexer, which can select the output from the computing CE and send it to the subsequent processing elements.

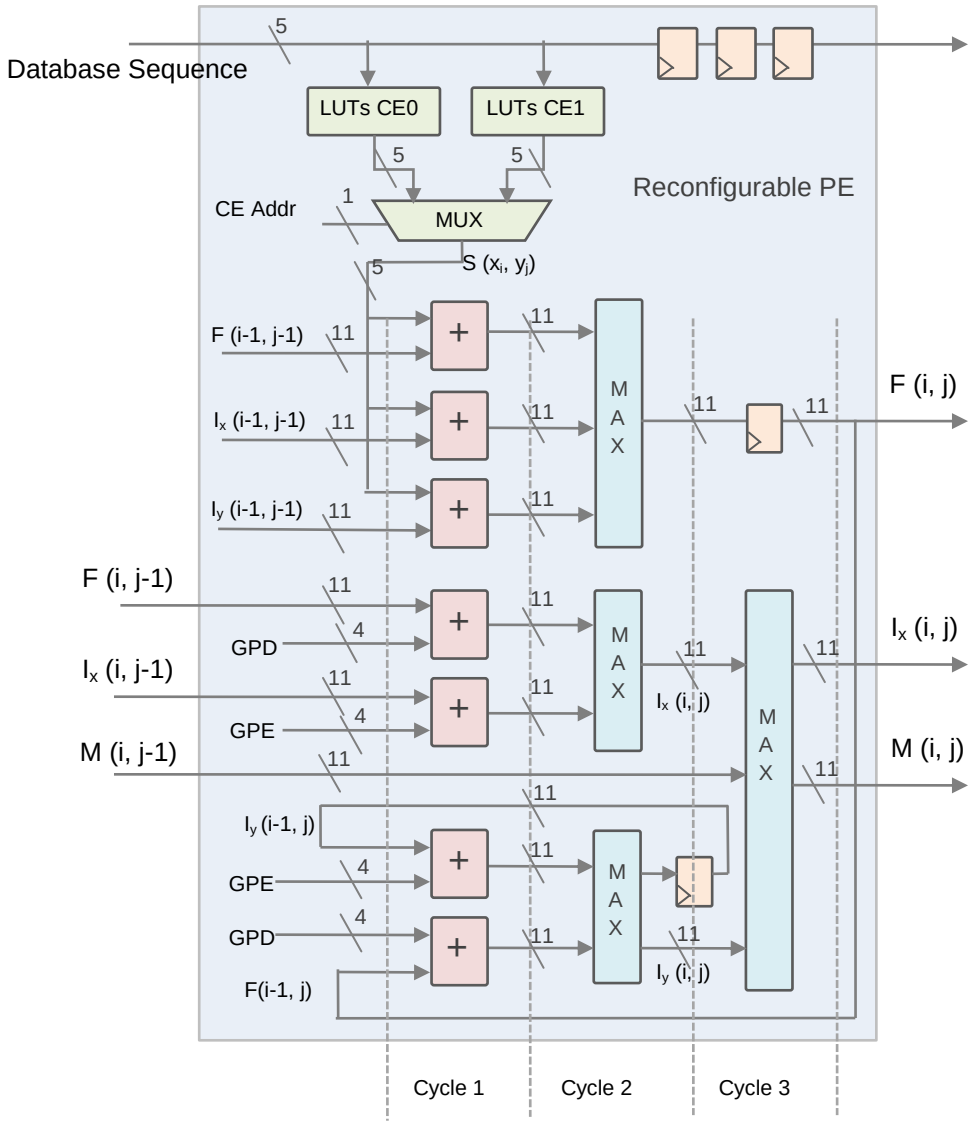


Figure 6.15 Reconfigurable PE architecture

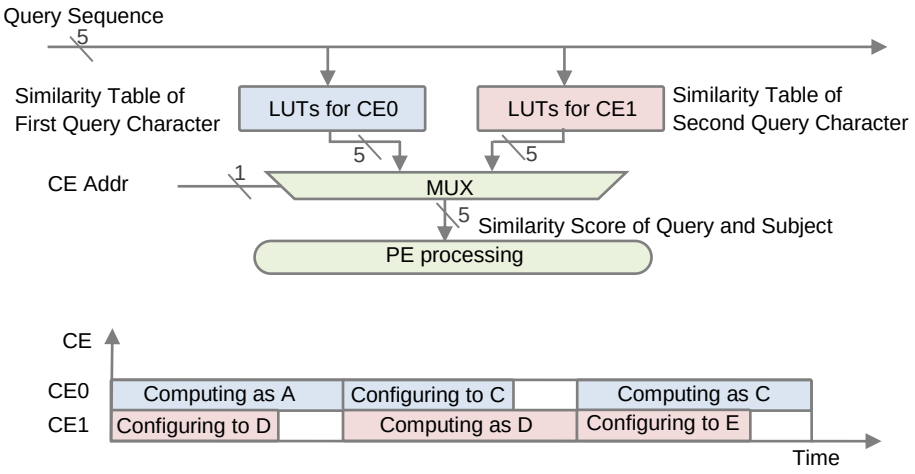


Figure 6.16 Two configuration elements used for PE folding

6.2.3 Integration with R3TOS

In the sequence alignment application, tasks are composed of a number of PEs, whose length and location can be flexibly defined to optimise the efficiency of resource usage. Figure 6.17 shows the task computing model for the sequence alignment application in both 2D and 3D views. In the proposed operating model, the R3TOS system is located at the static region, and all the SA application tasks can be flexibly placed anywhere in the reconfigurable region. The task size depends on the number of PEs, or the number of pipelined stages, which can also be flexibly customized depending on the currently available resources. Tasks having finished their computations can be removed from the chip in order to free up resources for

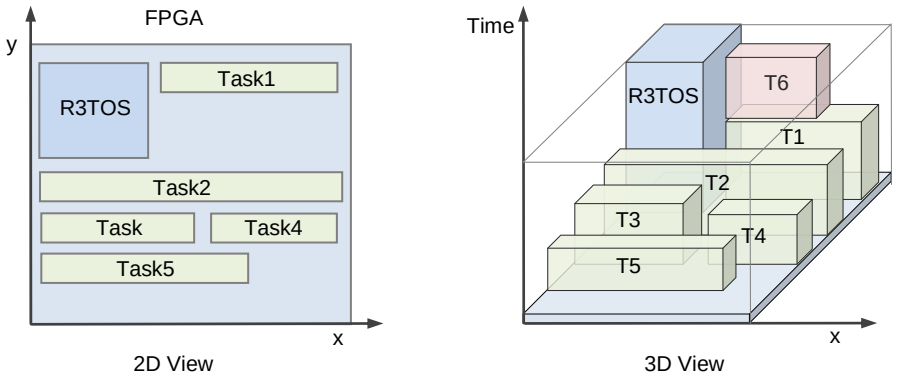


Figure 6.17 Task computing model for Sequence alignment application

later coming tasks; so for example task1 is removed to give space for task6 in this example.

Customising Tasks at Run-time

The most significant feature of this application is that the size of application tasks can be customised in response to the currently available resources. This is achieved by the task scheduler. In addition to the basic scheduling operations, the task scheduler can assemble an application task from a number of pre-generated single PEs, taking into the consideration the current maximum resources.

Figure 6.18 presents the proposed system diagram. The host API, which is the main CPU, is responsible for communicating with customers to deal with job requests. Since the main CPU is implemented on the MicroBlaze, which provides a number of standard communication ports, the communication between the R3TOS and the customer can be established through standard interfaces such as the TCP/IP to connect with the open network. After receiving a new request from the user, the main CPU then sends the new task to the task scheduler, together with the task performance requirements, such as the maximum folding number or the minimum PE number. Note that, in multi-user, multi-tasking applications, the performance of

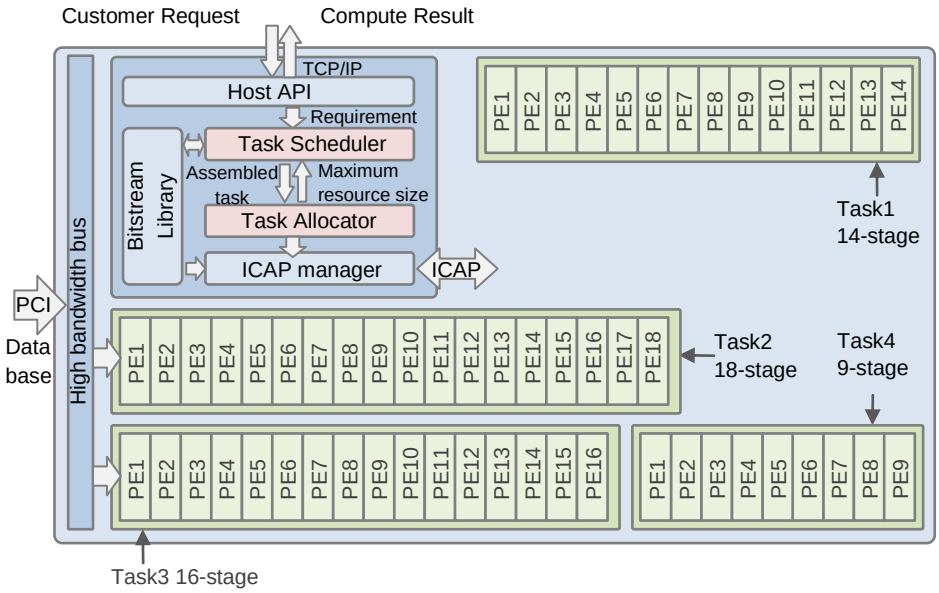


Figure 6.18 Proposed system operation diagram

each application task can vary depending on user requirements. For example, customers paying more can require more hardware resources for better performance.

After receiving the task and its requirements, the task scheduler assembles a task according to the user's requirements. The task is assembled by extracting and connecting the basic PEs, which are pre-synthesised off-line and stored in the bitstream library to be used for on-line generation. If the current resources are not sufficient for the assembled task, the scheduler will re-assemble the task by using the folding mechanism. The assembled task is passed to the task allocator, which can search for the best location and send it to the ICAP manager. After the position is decided, the allocator updates the maximum resource size and then feeds it back to the scheduler to be used for assembling the next task. Finally, the ICAP manager reads the bitstream of the assembled task from the bitstream library and downloads it to the configuration memory.

The database sequence is stored in an external off-chip memory, such as SRAM, since the on-chip memory is not sufficient to accommodate large database sequences. In order to overcome communication bottlenecks, such as the limited bandwidth to external memory, a PCI interface and a high-bandwidth bus is fixed in the static region to provide larger bandwidth. Tasks can be connected to the high bandwidth bus for faster data exchange whereas tasks placed elsewhere can communicate with the host using the ICAP. The high-bandwidth bus significantly reduces the communication overhead in applications requiring the exchange of large amount of data.

PE Interconnection

The application tasks are assembled from pre-synthesised PEs, which are separately synthesised off-line. In order to allow for on-line task generation, each PE has to be inter-connectable to its adjacent PEs. This requires the PEs to be specifically designed in a symmetrical and homogeneous way.

Inter-communication between the PEs is enabled by integrating an inter-PE Bus Macro (BM) during PE synthesis. BMs are hardware modules that have been

synthesised and manually placed and routed to use particular wires. When integrated with PEs, BMs can provide additional constraints to force PEs to use specified wires for both inputs and outputs. After integration with BMs, two PEs can communicate with each other as long as the they share the same wires, so that the output from the previous PE and the input to the next PE use the same wires at the same time.

BMs can specify their outputs to particular wires, by giving constraints on the configuration of Switch Boxes (SBs). SBs are programmable switches capable of connecting any wires passing through them. Figure 6.19 shows the interconnections of SBs in the chip network. In Xilinx FPGAs, each CLB, BRAM, DSP or IOB resource block is attached to a SB to connect to the chip network. The SBs can flexibly connect any of two or more wires by configuring its Programmable Interconnections Points (PIPs). Figure 6.20 shows all of the PIPs in one SB. Here, the *double lines* are used to connect adjacent SBs, while nonadjacent SBs can be connected by the *pent lines*. Both *double* and *pent* lines are short wires that give direct links between two or more SBs. The long wires include vertical long lines, horizontal long lines, and global lines, which cross the whole chip in different directions. Apart from the wires to other SBs, lines at the right-hand side are connected to its local resources in the same CLB, which includes data slices, control

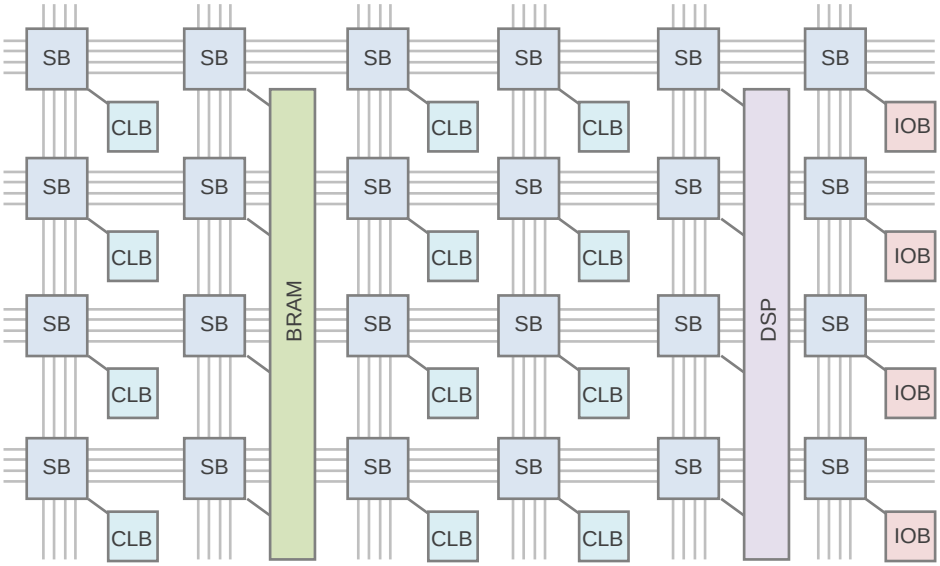


Figure 6.19 FPGA Switch Boxes (SB) and communication network

slices, and clocks. In the proposed implementation, since only two adjacent PEs require communication, the double lines are used to connect them. The top-left double lines are used for the inputs from the previous PE while the top-right double lines are used for the outputs to the next PE (see Figure 6.20).

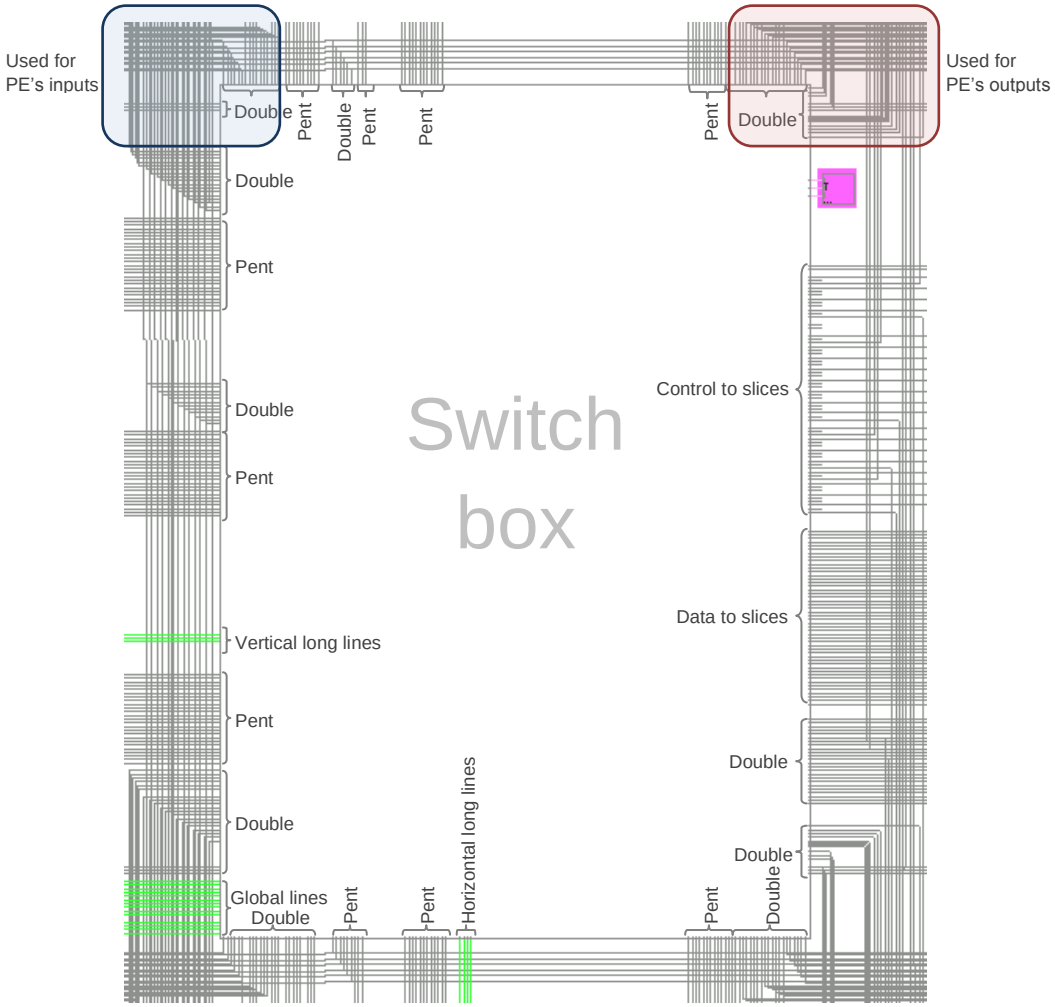


Figure 6.20 Interconnections of a switch box

Figure 6.21 gives an example of the low-level PE implementation and its interconnections. Each PE is area-constrained to two CLB columns, which can be mapped to any two adjacent CLB columns in one clock region. In this example, a SA task is composed of 6 PEs, which are allocated from left to right in the same clock region. In addition, a BRAM column is connected to the left of the array to temporally buffer the input data of database sequences. Note that the CLB columns

used for the PE array could be separated by other resource columns, such as BRAM and DSP columns, which intercept in the middle of the homogeneously aligned CLB columns. In such cases, those columns can be bypassed to maintain the homogeneity and continuity of the PE array. Data is processed from left to right by all the CLB columns (PEs) in a pipelined manner.

The interconnections between adjacent PEs are implemented using the horizontal short wires (double lines), which are shared between the first PE's outputs and the second PE's inputs. The double lines in the top-left corner are used as the inputs from the previous PE and the double lines in the top-right corner are used for the outputs to the next PE. Due to the symmetry between the double lines in the top-left and top-right corners, the previous PE's outputs can be directly connected to the inputs of the next PE (see Figure 6.20). In Virtex-5 family FPGAs, there are 6 double lines that directly and exclusively connect to the adjacent SB, giving a 6-bit bandwidth bus. When integrating PEs with BMs, the outputs from a PE can be constrained to use specified LUTs and flip-flops, whose outputs are tied to those particular wires (double lines) using specifically programed PIPs (see Figure 6.21). In the proposed design, the task requires 38 bits for its four inputs, including 11-bits for $F(i,j-1)$, 11-bits for $I_x(i,j-1)$, 11-bits for $M(i,j-1)$, and 5-bits for x_i, and 38 bits for its four outputs, comprising 11-bits for $F(i, j)$, 11-bits for $I_x(i, j)$, 11-bits for $M(i, j)$, and 5-bits for x_{i+1}. As a result, a bus bandwidth of at least 38 bits is required for both inputs and outputs. Since each SB can provide a 6-bit bandwidth, each PE's inputs occupy the left-hand 7 SBs for its inputs, which give a 42-bit bandwidth. Likewise, the right-hand 7 SBs together provide a 42-bit bandwidth used for the 38-bit outputs. In addition, a 5-bit width feedback link is integrated at the bottom of all the PEs, whereby the results produced by the last PE can be returned to the first BRAM column to give the input data for the next folding.

Despite the resource heterogeneity, all SBs are regularly distributed around the whole chip. As a result, the inter-PE buses can be more generically applied to the whole chip. Figure 6.22 a shows the allocation of the SBs of one clock region. The four different types of logic resources, IOBs, CLBs, BRAMs and DSPs, are circled,

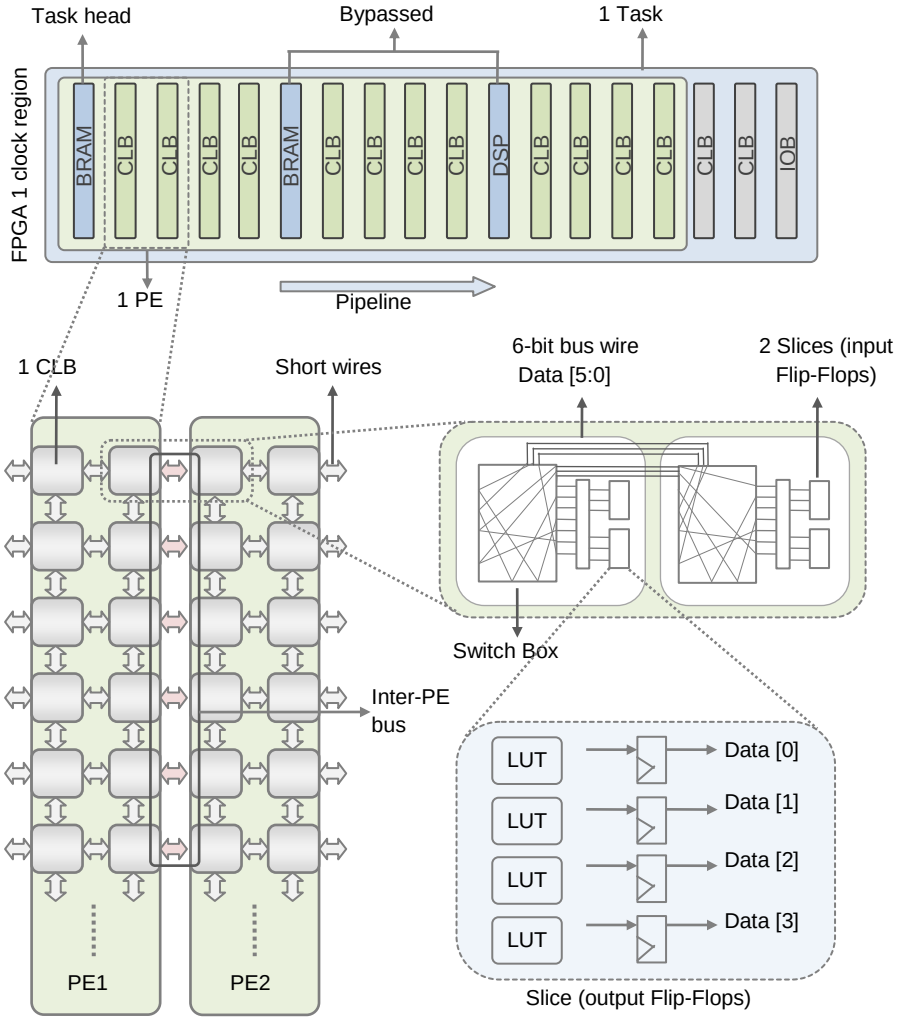


Figure 6.21 Inter-PE communication illustration

which shows that all SBs are aligned homogeneously regardless of differences in logic resources. The only differences between SBs are in their data lines to their local resource blocks. For example, in CLB blocks, the data lines are connected to SliceL and SliceM (see Figure 6.22 b); while in a BRAM block, the data lines are used as data and address buses to the memory block (see Figure 6.22 c). Likewise, the data lines of an SB in a DSP block are connected to the local DSP (see Figure 6.22.d); while an IOB's SB is connected to the chip IOs (Figure 6.22 e). Nevertheless, for all different types of resource blocks, the connections between their SBs and the chip network are all the same: all the short wires and long wires are uniformly connected to the network in the same way. Benefiting from above, the double lines can be used

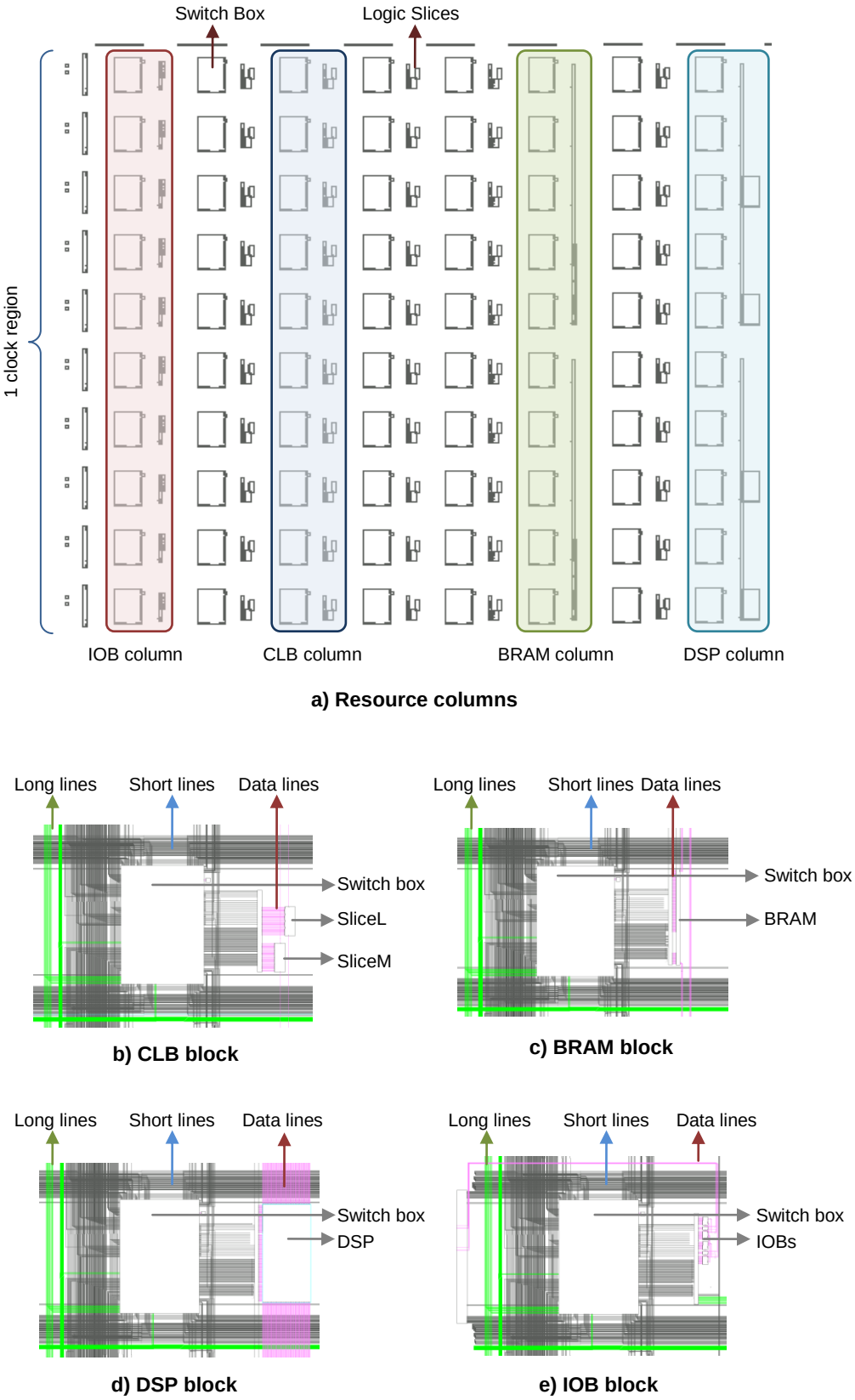


Figure 6.22 Homogeneously aligned Switch Boxes

more generically by any two resource columns. Moreover, the uniform distribution of SBs gives a constant propagation delay for all inter-PE communication, which makes task assembly easier in that timing violations are avoided.

Task Assembly

In the proposed operating environment, an application task consists of a task header, a task body, a task tail and several dummy columns.

The task header is implemented in a single BRAM column to store or buffer the input data before it is processed by PEs.

The task body is composed of a number of pipelined PEs. Depending on the requirements of the user and the current availability of resources, the length of the task body can be flexibly customised by the scheduler. During system operation time, the user can set upper and lower thresholds for task performance, for example in terms of the maximum and minimum number of PEs. Afterwards, the task scheduler firstly attempts to assemble a task aiming at maximum performance, where the number of PEs is chosen to be as large as possible so as to satisfy the user's requirement. If the available resources are insufficient to place the task, the task will then be folded to a reduced size. However if performance after folding would be reduced below the lower threshold set by the user, the task will wait in the task queue until any of the previously allocated tasks are removed from the chip.

After data is processed by the PE array, the result is outputted by the task tail, which can be implemented in either distributed RAMs or BRAMs depending on the size of the output data. In the present implementation, the result is fed back to the task header using the feedback link, which gives the input data for the next fold.

In order to deal with resource heterogeneity, dummy columns can be inserted to bypass one or more unused resource columns in one task. Since the CLB columns are intercepted by different types of resource columns, some of the PEs cannot be adjacently connected to each other. In such cases, dummy columns called column bypassers can be used to skip the heterogeneous resource column and pass the data

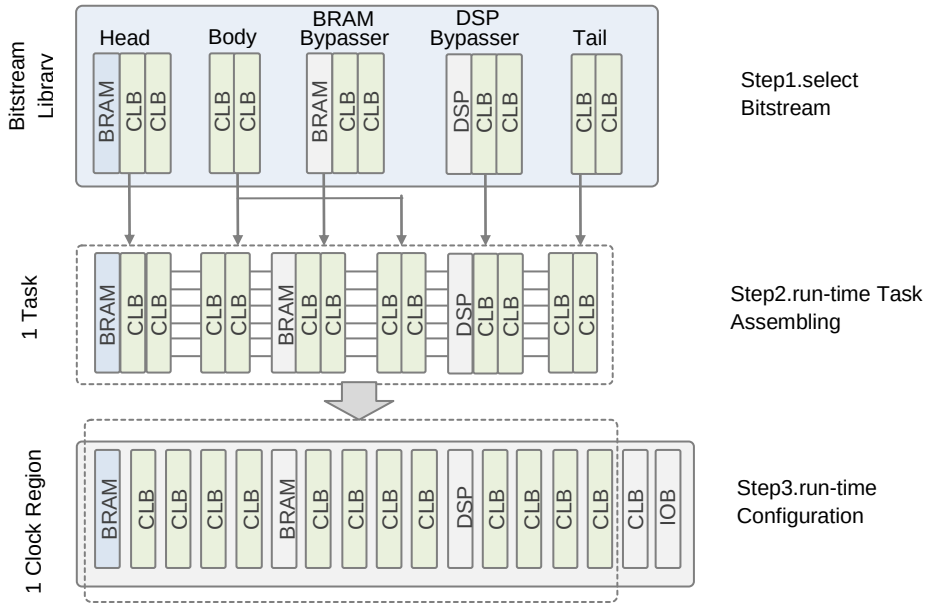


Figure 6.23 Steps to generate an application task

over the column without any data processing occurring. The dummy column is a separately synthesized hardware module in which the input SBs are directly linked to its outputs SBs, giving no latency on data transmission.

Figure 6.23 gives the steps used to generate an application task. Before the system starts, a number of hardware components have been implemented and their bitstreams are stored in the bitstream library. These include task headers, task bodies, task tails, and resource bypassers (dummy columns). During system operation time, depending on the requirements requested by the client user, the task scheduler can select appropriate components and assemble a fully functional application task. After the task is assembled, the task allocator will search for the best position and then pass it to the ICAP manager, which can configure the bitstream of the composed task to the configuration memory through the ICAP.

6.2.4 Multiple Frame Writing (MFW)

In order to configure a task to the chip silicon, the ICAP manager needs to write a number of configuration data sequences, which are called bitstreams, to the FPGA's configuration port, the ICAP.

Interfacing ICAP

The ICAP is a built-in bidirectional port which gives internal access to the configuration memory. More than one ICAP may be available on the same FPGA; for example Virtex-4 FX12 has two ICAPs located in the middle of the chip. However, they cannot be simultaneously used, and only one ICAP can ever be used at any one time. Figure 6.24 depicts the standard interface of an ICAP port, which includes a 32-bit input data port for configuration, a 32-bit output data port for reading back, a CE signal to enable the ICAP, a write enable signal for switching between W/R operations, and a busy signal for indicating if the ICAP is in its operational mode. The detailed operation timing graph have been published by the manufacturer [Xilinx2009] [Xilinx2012a].

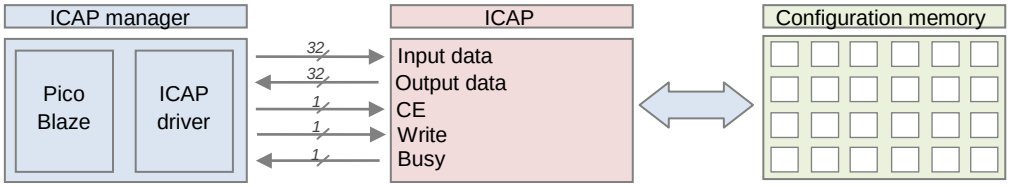


Figure 6.24 ICAP interface

According to the Xilinx user guide, the ICAP can be clocked up to 100MHz [Xilinx2009] [Xilinx2012a], and its data width can be changed from 8-bit to 32-bit. When operating in the 32-bit configuration mode at 100MHz, the ICAP provides a bandwidth of 32M bit /s, which is considerably higher than that of external configuration ports such as SelectMap and JTAG [Xilinx2012a].

However, the speed of configuration can be further increased by giving Multiple Frame Writing (MFW) instructions to the ICAP, which requires a change in the format of the commands in the bitstream. The following sections present the methods used to change the format of the bitstream to give MFW commands to the ICAP.

Accessing the configuration memory

In Xilinx FPGAs, a bitstream is composed of two sections: configuration commands and configuration frames. The configuration commands contain all the control information used to pre-set the ICAP before loading data to the configuration memory, including configuration modes and data size. The configuration frames are the content to be downloaded to the configuration memory, which describes all the logic of the hardware resources (see section 6.1.3).

TABLE 6.7 gives the bitstream used for writing two frames to the configuration memory operating in the normal configuration mode. The bitstream includes three parts: initial commands, frame data, and finishing commands. The initial commands take 12 clock cycles to send 12 32-bit words (an 8-hex value) to the ICAP, including synchronizing the ICAP, resetting the CRC code, setting the Frame Address Register (FAR), and defining the number of words to write. The word stored in the FAR indicates the physical address of the frame to be written. TABLE 6.8 lists the representations of bits in the FAR register, in which bit22 indicates the top or bottom half of the FPGA; bit21-bit19 determine the type of resources (000 for CLB, 001 for BRAM connection, and 010 for BRAM content); bit18-bit14 specify the address of the clock region (row address); bit 13-bit6 present the column address, and bit5-bit0 give the frame address of the column. In addition, NO-OP operations are inserted to give the ICAP more time redundancy to execute certain commands. At the end of the initial commands, two commands, type-1 Frame Data Register Input (FDRI) and Type-2 FDRI, are used to specify the number of frame words to be written. As soon as the FDRI register is written with the number of frames, the frame data is sequentially sent to the ICAP from the next clock cycle. In this example, two frames require 82 32-bit words, which takes 82 clock cycles to complete. After all of the frame words are sent to the ICAP, a sequence of finishing commands is used to reset the ICAP to its initial stage for its next round of operation. Likewise, the bitstream for the readback operation is similar to that of the writing operation, which is composed of initial commands, frame data, and finishing commands (see TABLE 6.9). The initial commands are different in that the number of frame words is written to the Frame Data Register Output (FDRO) instead of the FDRI. In addition, the initial

TABLE 6.7 BITSTREAM FOR WRITING TO CONFIGURATION MEMORY (NORMAL MODE)

	Cycle	Read/ write	Configuration data	Explanation
Initial Commands	1	W	20000000	NO-OP
	2	W	AA995566	Synchronization command
	3	W	30008001	Type 1 write 1 words to CMD
	4	W	00000007	Reset CRC
	5	W	20000000	NO-OP
	6	W	20000000	NO-OP
	7	W	30008001	Type 1 write 1 word to CMD
	8	W	00000001	WCFG command
	9	W	30002001	Type 1 write 1 word to FAR
	10	W	00000000	FAR=00000000
	11	W	30004000	Type 1 write 0 words to FDRI
	12	W	00000029	Type 2 write 41 words to FDRI
Frame data	13	W	xxxxxxxx	Write word 1 (frame1)
	14	W	xxxxxxxx	Write word 2 (frame1)
	15-52	W	...	Write word 3-40 (frame1)
	53	W	xxxxxxxx	Write word 41 (frame1)
	54	W	xxxxxxxx	Write word 42 (frame2)
	55	W	xxxxxxxx	Write word 43 (frame2)
	56-93	W	...	Write word 44-81 (frame2)
	94	W	xxxxxxxx	Write word 82 (frame2)
Finishing Commands	95	W	20000000	NO-OP
	96	W	30008001	Type 1 write 1 word to CMD
	97	W	00000005	START command
	98	W	30008001	Type 1 write 1 word to CMD
	99	W	00000007	Reset CRC
	100	W	30008001	Type 1 write 1 word to CMD
	101	W	0000000D	DESYNC command
	102	W	20000000	NO-OP
	103	W	20000000	NO-OP

commands contain a SHUTDOWN operation to protect the configuration memory during readback operations. In TABLE 6.7, the frame words are written to the configuration memory in the sequential mode. In such cases, the 41 words of each

frame require 41 clock cycles, and the total configuration time can be calculated by Equation 6.4.

$$T_{cfg} = T_{init} + (41 \times N_{frame}) \times f_{clk} + T_{fin}$$ **Equation 6-4 Normal configuration**

T_{cfg} is the total configuration time, which is the sum of the time for sending initial commands (T_{init}), the time for the configuration of frame data, and the time for finishing commands (T_{fin}). The initial and finishing commands require fixed time durations, which are insignificant compared with that for frame data configuration. The time needed for configuring the frame data is the product of the number of frame words (*41*) and the number of frames to be configured (N_{frame}), multiplied by the frequency of the system clock (f_{clk}).

TABLE 6.8 FRAME ADDRESS REGISTER

Bit	Width	Function
31-23	9	Reserved
22	1	Top/bottom
21-19	3	Block type
18-14	5	Row address
13-6	8	Column address
5-0	6	Minor address

Writing multiple frames using compressed bitstream

However, the configuration speed can be increased at least twice if identical frames exist in the same bitstream. To replicate one identical frame to a different frame address, only 2 more cycles are required. The replication saves 39 clock cycles in configuring one frame compared with conventional bitstream configuration which requires 41 cycles for one frame configuration. The time needed to configure identical frames can be expressed by Equation 6.5:

$$T_{cfg} = T_{init} + (41 + 2 \times N_{frame}) \times f_{clk}$$ **Equation 6-5 MFW configuration**

The replication of identical frames is also called MFW, and a bitstream using such

TABLE 6.9 BITSTREAM USED FOR READING BACK FROM THE CONFIGURATION MEMORY

	Cycle	Read/ write	Configuration data	Explanation
Initial Commands	1	W	20000000	NO-OP
	2	W	AA995566	Synchronization command
	3	W	30008001	Type 1 write 1 words to CMD
	4	W	00000007	Reset CRC
	5	W	20000000	NO-OP
	6	W	20000000	NO-OP
	7	W	30008001	Type 1 write 1 word to CMD
	8	W	0000000B	SHUTDOWN command
	9	W	20000000	NO-OP
	10	W	20000000	NO-OP
	11	W	20000000	NO-OP
	12	W	20000000	NO-OP
	13	W	30008001	Type 1 write 1 word to CMD
	14	W	00000004	RCFG command
	15	W	30002001	Type 1 write 1 word to FAR
	16	W	00000000	FAR = 00000000
	17	W	28006000	Type 1 read 0 words from FDRO
	18	W	00000052	Type 2 read 82 words (2 frames)
	19	W	20000000	NO-OP
	20	W	20000000	NO-OP
Frames	21	R	xxxxxxxx	Read word 1
	22	R	xxxxxxxx	Read word 2
	23-101	R	...	Read word 3-81
	102	R	xxxxxxxx	Read word 82
Finishing Commands	103	W	20000000	NO-OP
	104	W	30008001	Type 1 write 1 word to CMD
	105	W	00000005	START command
	106	W	30008001	Type 1 write 1 word to CMD
	107	W	00000007	Reset CRC
	108	W	30008001	Type 1 write 1 word to CMD
	109	W	0000000D	DESYNC command
	110	W	20000000	NO-OP
	111	W	20000000	NO-OP

commands is named a compressed bitstream. The compressed bitstream was originally developed by Xilinx, in aiming to compress the size of a bitstream so that

it could be easily stored in external non-volatile memories such as FLASH memories. On the other hand, the compressed bitstream can also significantly reduce the configuration overhead when multiple identical frames are reconfigured at run-time. In the light of the above, this technique is applied in the proposed system to speed up dynamic reconfiguration, since a considerable number of identical PEs are heavily reused in highly pipelined PE arrays. Moreover, not only can the multiple PEs be replicated at faster speed, but also the finished tasks can be blanked in a shorter time, as the blanking operation is equivalent to replicating zero frames to the whole task region. When configuring multiple identical frames, the compressed bitstream can achieve up to 20 times speed-up in comparison to normal sequence-based configuration.

Figure 6.25 gives the comparison of normal configuration and MFW configuration. In the normal configuration mode, the 41 words need to be rewritten for each frame, even if two frames are identical. However in the MFW configuration mode, the same 41 words can be copied to different frames by only updating the frame address in the FAR. As a result, the MFW configuration requires only 2 clock cycles to configure an identical frame, whereas the normal configuration requires 41 cycles.

TABLE 6.10 gives the bitstream used for configuring two identical frames to the configuration memory using MFW mode. The initial and finishing commands are the same as those in normal configuration. However, when configuring the second frame, MFW only needs to update the address in the FAR, rather than rewriting all of

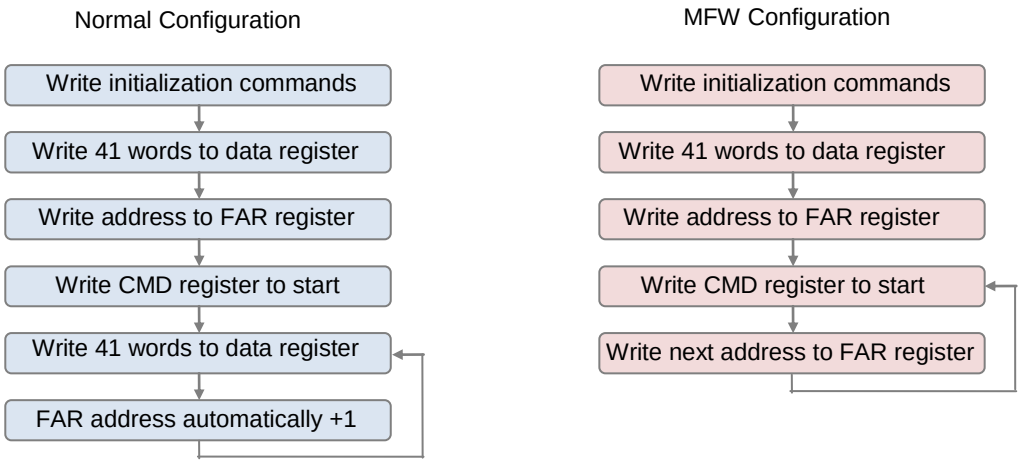


Figure 6.25 Steps for normal and MFW configuration

TABLE 6.10 BITSTREAM FOR WRITING TO CONFIGURATION MEMORY (MFW MODE)

	Cycle	Read/ write	Configuration data	Explanation
Initial Commands	1	W	20000000	NO-OP
	2	W	AA995566	Synchronization command
	3	W	30008001	Type 1 write 1 words to CMD
	4	W	00000007	Reset CRC
	5	W	20000000	NO-OP
	6	W	20000000	NO-OP
	7	W	30008001	Type 1 write 1 word to CMD
	8	W	00000001	WCFG command
Frames	9	W	xxxxxxxx	Write word 1
	10	W	xxxxxxxx	Write word 2
	11-48	W	...	Write word 3-40
	49	W	xxxxxxxx	Write word 41
Finishing Commands	50	W	30002001	Type 1 write 1 word to FAR
	51	W	00000000	FAR=00000000
	52	W	30014000	Multiple frame write
	53	W	30002001	Type 1 write 1 word to FAR
	54	W	00000001	FAR=00000001
	55	W	30014000	Multiple frame write
	56	W	20000000	NO-OP
	57	W	30008001	Type 1 write 1 word to CMD
	58	W	00000005	START command
	59	W	30008001	Type 1 write 1 word to CMD
	60	W	00000007	Reset CRC
	61	W	30008001	Type 1 write 1 word to CMD
	62	W	0000000D	DESYNC command
	63	W	20000000	NO-OP
	64	W	20000000	NO-OP

the 41 frame words. As a result, the entire configuration requires 64 clock cycles, which is just over one half of the number used in normal configuration mode (103 cycles). Note that, when more identical frames are required, the configuration speed increases dramatically compared to normal configuration, with up to 20x speed-up.

6.2.5 Tests and Results

The proposed sequence alignment application was implemented on an XC5VLX110T FPGA on an Alpha Data ADM-XRC-5LX board. Figure 6.26 gives the layout of the whole chip implemented with the R3TOS and two SA application tasks. The static region is fixed in the left half of the 5th clock region, where both the R3TOS and ICAP are located. The other areas are used as the reconfigurable region, in which application tasks can be flexibly placed at arbitrary positions.

During the system operation time, multiple tasks can execute in parallel in different positions at the same time, and their size and performance can be flexibly customized depending on both of the user requirement and the currently available resources. For example in Figure 6.26, two SA tasks with different performances levels run in parallel for two different query sequences. The first task is composed of 21 PEs using 42 CLB columns, whereas the second task consists of 9 PEs that occupy 18 columns.

In the proposed SA application, the task head is implemented in a BRAM column, which temporarily buffers the input data of database sequences from the external memory. In the target Virtex-5 LX110T FPGA chip (XC5VLX110T), there are 8 clock regions, and each contains 4 BRAM columns, so that there are in total 8×4 BRAM columns available to be used by the task head, giving 32 possible locations at which to place an SA task. In addition, in order to increase the communication throughput, a high-bandwidth bus is fixed in the leftmost region of the chip, whereby the content of the first BRAM column can be quickly updated with low time and area overheads. On the other side, the ICAP communication mechanism is used to read the result outputted by the task tail, since the size of the result (alignment score) is relatively small. In the target chip, each clock region contains 50 CLB columns; hence there are a total of 50×8 CLB columns available in the whole chip, allowing for a maximum of 8 fully pipelined tasks (with 25 PEs) running at the same time. If the required number of PEs for one task is more than 25, it will not fit within one clock region. In such a case, the PE array can be routed across multiple clock regions by applying the snake strategy [Iturbe2011a], so that one SA task can have a pipeline with a maximum length of 200 PEs.

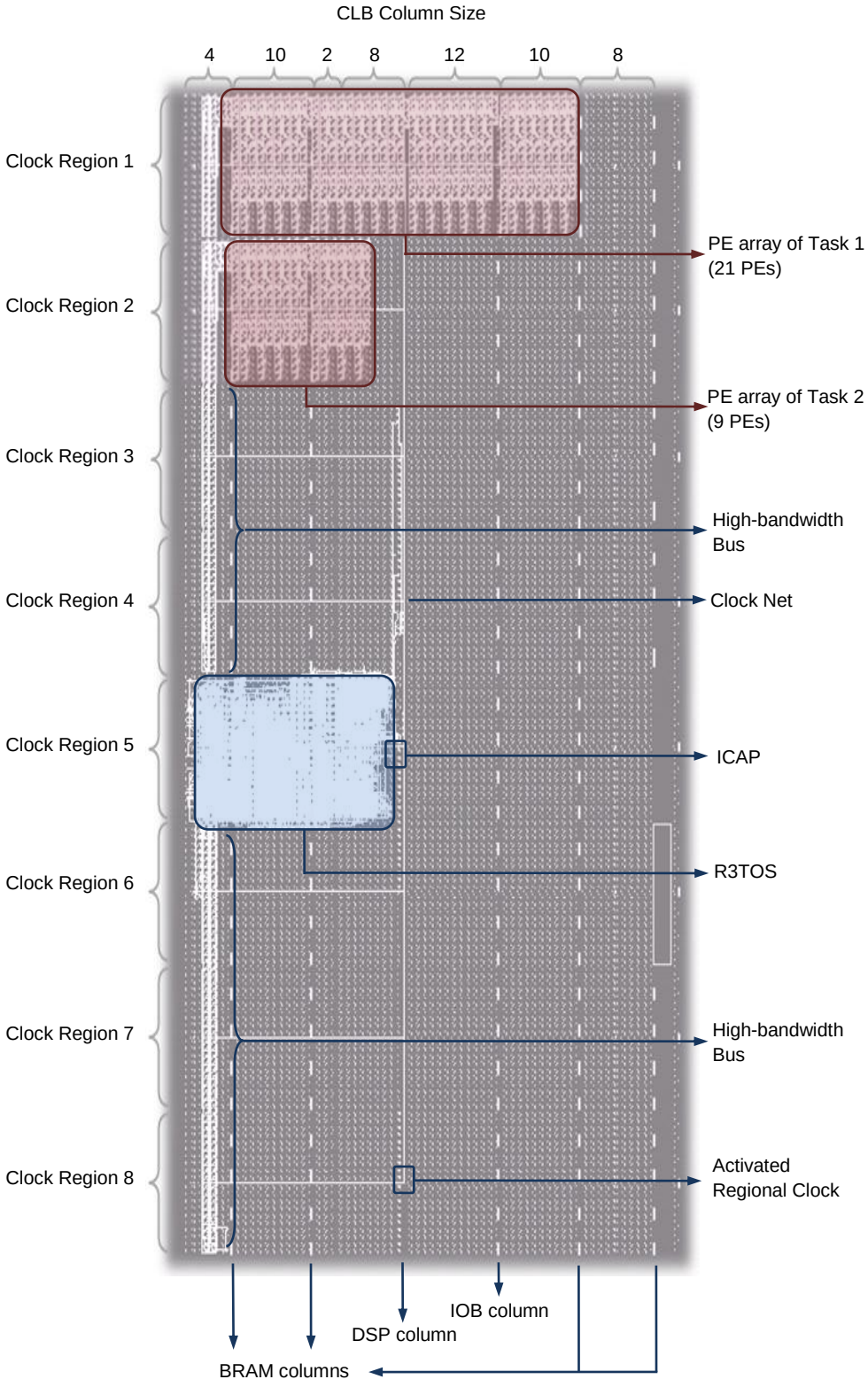


Figure 6.26 Implemented R3TOS and pairwise sequence alignment cores on an XC5VLX110T FPGA

Figure 6.27 illustrates the schematic implementation of a single PE and its interconnections to adjacent PEs. As mentioned in Section 6.2.3, each PE is implemented in two CLB columns, and requires 38-bit input and 38-bit output, which are implemented using 7 SBs each on the left and right sides. In each SB, there are 6 double lines directly connected to the next SB, allowing 6-bit data to be transmitted directly between PEs. The SB at the bottom of a PE is used to build the direct link to feedback the results from the task tail to the task head. The feedback link provides a bus horizontally connecting the task tail to the task head, whereby data can be fed back to the input buffer for the next folding.

To test the performance of the PEs, UniProtKB/ TrEMBL [Isa2012] database sequences were used as the task inputs, and the results compares with those of previous approaches implemented in both software and hardware. The results show that the proposed reconfigurable PE array has achieved significant speed-up compared with previous approaches, and the dynamic run-time task assembly gives zero penalties for the performance of all PEs [Isa2012, Hong2013b].

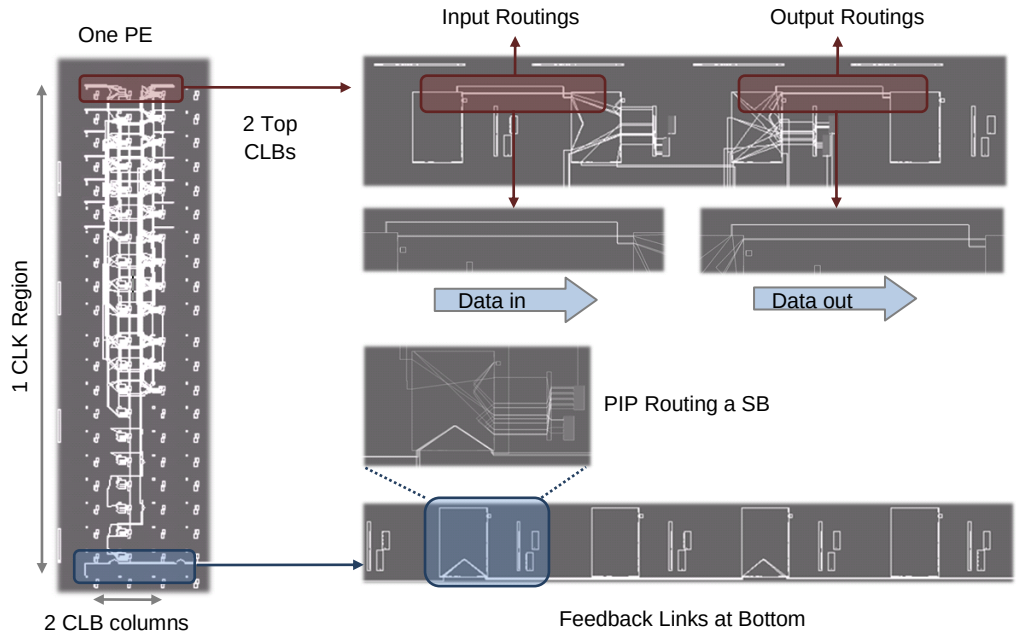


Figure 6.27 Implemented PE and communication routing

TABLE 6.11 gives a comparison of the software approach and the hardware implementation. The results show that, when using 25 pipeline stages, which is the maximum length in one clock region, the hardware implementation increases computing speed by ~30x, in comparison with the performance of SSEARCH35 [Isa2012] software running on an Intel(R) Quad Core 64-bit Q8300 CPU, [Hong2013b].

TABLE 6.11 PE EXECUTION TIME COMPARED WITH SOFTWARE

Query Accession	Length	PE	Fold	Execution Time (s)		Speed-up
				Ours	SSEARCH	
P02652	100	25	4	303	9416	×31.08
Q9H3V2	200	25	8	599	17160	×28.65
Q8NC42	400	25	16	1215	35992	×29.62

PE performance is also compared with previously proposed hardware implementations. Note that the performance of the hardware implementations not only depends on the design, but is also closely related to the device technology used, such as gate delay and propagation delay. Taking into the consideration device performance, the speed-up can be normalized as:

$$SpeedUp_{normalized} = \frac{SpeedUp_{Raw}}{Number_{LC}} \times Device\,Delay$$ **Equation 6-6 Normalized speed-up**

in which $SpeedUp_{Raw}$ is the speed improvement obtained from the real test, $Number_{LC}$ is the number of logic cells consumed by each PE, representing area consumption, and *Device Delay* is the device propagation delay.

TABLE 6.12 presents the test results of the proposed PEs, which achieve 1.53x and 2.4x normalized speed-up in comparison with results reported elsewhere [Oliver2005, Benkrid2009].

TABLE 6.12 PE SPEED-UP COMPARED WITH HARDWARE

Reference	Device	Resource Ratio[1]	Device Delay Ratio[2]	Raw Speed-up	Normalized Speed-up
[Benkrid2009]	XC2V6000	0.69	0.23	×4.58	×1.53
[Oliver2005]	XC2V6000	0.49	0.23	×5.13	×2.40

[1] Resource Ratio = LCs consumed by our approach / LCs consumed by Reference.
[2] Device Delay Ratio = XC5VLX110 delay / XC2V6000 delay

TABLE 6.13 gives the evaluation of system configuration speed. On average, the system requires ~18 μs to find the best location for a task. To place an application task, the system needs to configure the task header, the task body, and the task tail to the target region, which requires less than 150 μs for a task of average size. Note that, when configuring multiple identical frames, the configuration speed is significantly improved by MFW. For example, when configuring 25 PEs, the time needed for configuring each PE is reduced to 134.22/25 = ~5.4 μs, which is 13 times faster than configuring a single PE task. Likewise, when removing a finished task from the chip, MFW can be applied to reduce the configuration time to less than 40 μs.

In terms of area consumption, the R3TOS implemented on Virtex-5 FPGA requires fewer resources than its implementation on Virtex-4 FPGA. When implemented on Virtex-5, the whole system requires 1210 slices and 14 BRAMs, which is only 7% and 5% of the total resources of the chip.

TABLE 6.13 TASK PLACER PERFORMANCE (UNDER 100MHZ)

Process	Time
Finding Location (Average Time)	18 μs
Configure PE Header	68.38 μs
Configure 1 PE Body	2.98 μs
Configure 1 PE Tail	2.98 μs
Configure whole task (10x PE)	111.18μs
Configure whole task (25x PE)	134.22μs
Update Data Input (Configure one BRAM content)	26.56 μs
Blank whole task (25x PE) (Power Saving)	36.32μs

On-line task customisation aims to improve overall system performance by more efficiently using hardware logic. In order to evaluate the improvement in overall system performance, a real-time, multi-user, multi-tasking environment is emulated, in which a number of tasks with different ranges of user requirements can be customised (folded) to fit into currently available resources. The results show that the overall task finishing rate is significantly improved when using folding adjustment. TABLE 6.14 presents 6 task sets (φ1- φ6) with different ranges of user requirements varying from low to high, and the results for task finishing rates. Each of the first three sets (φ1- φ3) contains 100 tasks, which are requested every 36 seconds. Likewise, each of the second three sets (φ4- φ6) has 200 tasks sent every 18 seconds. Every task set is tested for 60 minutes on XC5VLX110 (with 50 × 8 CLB columns) using the same query accession (P02652). The performance requirement is represented as its maximal allowable fold. For instance, a task for which higher performance is required will have less maximal allowable fold, which prevents the task from being folded too many times. In contrast, a task for which lower performance is required will have a higher maximum allowable fold, whereby the task can be folded more times to reduce its size at the cost of performance. The maximal allowable fold is a random number ranging from $Fold_{min}$ to $Fold_{max}$. For example, when testing the first task set (φ1), a task is requested by the user every 36 seconds, and the maximal allowable fold is a random number ranging from 4 to 32. The system will firstly attempt to place the task with its minimal fold of 4. If the current resources are insufficient to place the task with 4 folds, the system will keep doubling the fold number until either the task can fit into the chip, or its maximum fold number is reached. The folded task will be configured to the chip if its folding number is less than its maximum fold number; otherwise the task has to be queued in the ready task list to wait for existing tasks to be removed. At the end of the test, the number of finished tasks ($N_{finished\_tasks}$) and the task completion rate ($R_{finished\_tasks}$) are obtained. The results show that, when using folding adjustment, the overall task completion rate is improved by ~2.5x compared with tasks without using folding adjustment (see Figure 6.28). This improvement becomes more significant when more flexible task requirements are applied (~3x), since tasks can be folded more

times to fit into smaller resource slots, whereby resources are more efficiently used rather than staying idle.

TABLE 6.14 SYSTEM PERFORMANCE TEST

Task set	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6
$Fold_{max}$	32	16	8	32	16	8
$Fold_{min}$	4	4	4	4	4	4
$N_{requested\_tasks}$	100	100	100	200	200	200
$R_{requested\_tasks}$	Every 36 s			Every 18 s		
Using folding adjustment						
$N_{finished\_tasks}$	91	73	57	93	78	59
$R_{finished\_tasks}$	91%	73%	57%	46.5%	39%	29.5%
Not using folding adjustment						
$N_{finished\_tasks}$	32	32	32	38	38	38
$R_{finished\_tasks}$	32%	32%	32%	19%	19%	19%

Chip size: 50 × 8 CLB columns, Test duration: 1 hour (3600s)
Query accession: P02652, Folding starting number: 4

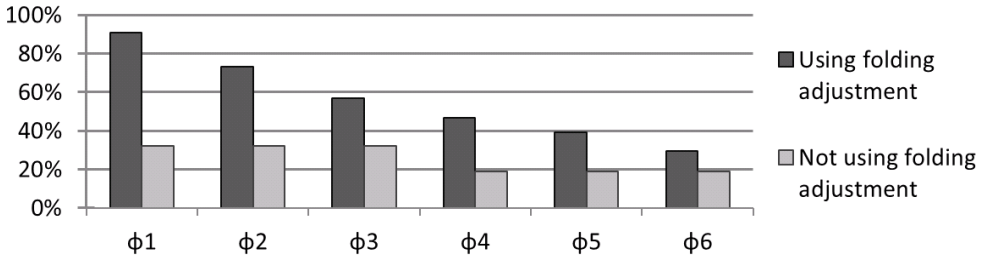


Figure 6.28 Performance improvement by using folding adjustment

6.3 Conclusion

This chapter presented the development of two applications using the R3TOS, the K-NN classifier and pairwise biological SA. The former is a classification algorithm used in bioinformatics to compute the distance between a query vector and vectors in a training set. The latter is a dynamic programing-based algorithm used for identifying the regional/global similarities between two biological sequences.

The two applications were used to demonstrate R3TOS in different respects. In the K-NN application, the system shows its capability of allocating a task (a K-NN core) at an arbitrary position, and communicating with tasks using the ICAP mechanism. In addition, the system's fault tolerance is improved by reallocating a task from

damaged resources. On the other hand, the SA application demonstrates that the system is capable of adjusting the size of an application task (an SA PE array) in response to the currently available resources, whereby the overall system performance is improved by making better use of idle hardware resources. Moreover, this chapter gave a more detailed illustration of low-level system operation in the context of two case studies, including the mapping between bitstreams in the configuration memory and their corresponding functionalities in the logic layer, as well as of low-level methods such as MFW to improve configuration speed,.

The results show that the proposed R3TOS system has achieved a higher flexibility in managing hardware resources in multi-user, multi-tasking applications, whereby hardware tasks can be swapped in/out from the chip in a time-multiplexed fashion. Benefitting from this, the efficiency of resource usage and overall system performance are improved by sharing limited hardware resources between different tasks and users.

Chapter 7

Conclusion and Future Work

This thesis presented a Reliable, Reconfigurable, Real-Time Operating System, the R3TOS, which bridges the widening gap between the development of complex high-level user applications and their implementation in complicated low-level Dynamic Partial Reconfiguration (DPR) hardware. The R3TOS significantly exploits up-to-date DPR technologies, while providing a generic software-like programming API to application users whereby software-centric application developers can benefit from DPR capabilities such as high performance, low power, and flexible adaptability.

This chapter firstly summarises and draws conclusions about the main achievements of each chapter. The impact of the work on some major objectives in this field is then evaluated. Finally, some criticisms of this system are discussed, which should assist in future work related to this research.

7.1 Summary and Conclusions

The R3TOS is developed with three major goals in mind of high reliability, flexible reconfigurability, and improved real-time performance, by means of the use of advanced DPR techniques. A novel system model and innovative scheduling and allocating algorithms have been proposed, and implemented with real hardware logic. The system has been demonstrated in the context of two applications.

TABLE 7.1 describes the main achievements reported in Chapters 3 to 6, after an introduction and review of relevant work were given in Chapters 1 and 2 of this thesis.

In chapter 3, the overall system was introduced and illustrated with regard to its three objectives of reliability, reconfigurability, and real-time performance. Reliability is improved by isolating faults within each module, triplicating reconfigurable regions, and applying the scrubbing technique utilizing ECC mechanisms. In terms of reconfigurable flexibility, the system allows tasks to be arbitrarily placed anywhere on the chip, with the support of four communication mechanisms and efficacious allocation algorithms. In addition, real-time performance is guaranteed by integrating the proposed novel scheduling algorithms with commercially used ROSs such as FreeRTOS.

Chapter 4 presented one efficient real-time scheduling algorithm (FAEDF) and two efficacious allocation algorithms (EAC and EAV). Compared with previous approaches, the FAEDF improves scheduling efficiency up to two fold, while an increase of ~25% allocation efficiency is achieved using EAC. Moreover, when combining the FAEDF and EAV, the task finishing rate is further improved up to two fold.

Chapter 5 presented the hardware implementation of the R3TOS HWμK, which includes the task scheduler, the task allocator, and the ICAP manager. The three components are implemented on three fault-tolerant microprocessors, which utilise ECC mechanisms to recover from any faults with low overheads. In addition, a SMP-based architecture is implemented to allow shared memory based programming in the R3TOS.

Finally, in chapter 6, the R3TOS was demonstrated in the context of two applications, namely K-Nearest Neighbour Classifier (K-NN) and Sequence Algiment (SA). The former proves the basic concept of R3TOS that tasks can be reallocated flexibly from damaged resources and can communicate without

TABLE 7.1 CONCLUSION OF MAIN ACHIEVEMENTS

Chapter	Subjects	Novel Contributions
Chapter 3 System Overall	Reliability	• Tasks are logically and operationally isolated to prevent faults from propagating to other units. • Reconfigurable regions are triplicated to give regional redundancies. • Faults can be detected and fixed by ECC mechanism and scrubbing technique.
	Reconfigurability	• Tasks can be flexibly placed at arbitrary positions on the chip. • Four novel communication mechanisms are developed to enable inter-task communications. • Novel allocating algorithms are developed to reduce chip fragmentation.
	Real-time	• Novel real-time scheduling algorithms are developed to give hardware real-time scheduling • FreeRTOS software kernel is integrated to provide software tasks with real-time performance.
Chapter 4 Algorithms Design	Scheduling algorithm FAEDF	• FAEDF achieves up to 2x scheduling efficiency compared with conventional EDF algorithm.
	Allocating algorithm EAC, EVC	• EAC achieves ~25% improvement on allocating efficiency compared with MER and VLS algorithms. • EVC-FAEDF achieves up to 2x task finishing rate compared with EVA-EDF.
Chapter 5 Hardware Implementation	Fault tolerant microprocessor	• EPA is developed to enable processor's program memory to be protected by ECC memories. • Processor can automatically self-recover from any fault. • Errors can be detected and fixed with minimal time and area overheads.
	R3TOS HWμK	• Hardware tasks are effectively scheduled, and hardware resources are efficiently managed with low time and area overheads. • Two orders of improvement on configuration speed are achieved compared with Xilinx ICAP.
	SMP communication mechanism	• The mechanism allows shared memory based programing in R3TOS. • Task can communicate each other with low time and area overheads.
Chapter 6 Application Development	K-NN Classifier	• Tasks can communicate without on-chip routing • Fault tolerance is improved by task reallocating. • Configuration speed is improved with low area overhead.
	Sequence Alignment	• Tasks can be customised at run-time in response to the current available resources. • MFW increases configuration speed by ~20x. • PE achieves ~30x and ~2x speed-up compared with previous software and hardware approaches respectively • The overall system performance is improved by ~2.5x.

conventional routings. The latter application demonstrates that the system is capable of customising tasks at run-time in response to both user requirements and the currently available resources, whereby the high performance of PEs is maintained and overall system performance is improved by ~2.5x.

Finally, it is acknowledged that work completed in 2012 on a previous version of the R3TOS has been published by group member Xabier Iturbe in his doctoral thesis [Iturbe2013c], That publication does not include work completed in the last year, such as on the SMP and two bioinformatics applications. Moreover, some additional work has been published on reliable clock-nets [Iturbe2012], a power-chain application [Iturbe2013b], and an SDR application [Torrego2013]. Last, but not least, I acknowledge that some of the ideas and results in this thesis have been published in our journal and conference papers (see relevant publications, page viii). Nevertheless, all the contents of this thesis, including the text and diagrams are completely original and no diagram or sentence has been copied from any previous publications.

7.2 Novelty contributions

The novelty contributions of this work are: 1) a novel implementation of an efficient task scheduler and a task allocator, which are the main components of the system. 2) The implementation of a scheduling algorithm (FAEDF) and two allocating algorithms (EAC and EVC), which are used together and have improved the task finishing rate by up to two times. 3) A novel implementation of a fault-tolerant microprocessor, which is capable of automatically correcting a single error without halting the processor operating, as well as recovering from multiple faults by using the scrubbing technique. 4) A novel symmetric multiprocessing (SMP)-based architectures that allows the system to be programed in through a shared memory-like interface, with relatively less area and time overheads. 5) Two application demonstrations, namely the K-NN and the SA application, whose results show that the system has significantly improved its operating performance towards both area and time efficiency, with ~2x improvement compared with other non-reconfigurable systems.

7.3 Evaluation of Impacts

The R3TOS has achieved high improvement in reliability, reconfigurability, and real-time performance by means of advanced DPR techniques. This has a positive impact in terms of satisfying major objectives in this field. The following sections describe some of the main achievements of this work.

Reliable Computing

Reliable computing is demanded in a number of safety critical applications, such as aerospace where devices have to be designed to counter highly radiative environments. Reconfigurable hardware, such as the FPGA, is preferred since it allows for in-system reconfiguration after launch. The R3TOS has significantly improved the reliability of reconfigurable hardware, from its computing model to its hardware implementation, making it more likely to meet safety-critical criteria.

In Chapter 3, a new task dependency model are proposed, where each task is both physically and operationally isolated, so that a fault occurring in the silicon can be self-contained within one unit or task module, rather than being propagated to other units. In addition, the R3TOS partitions the whole chip region into three FCCRs to provide regional redundancy. In Chapter 5, the low-level implementation of a proposed fault-tolerant microprocessor is presented. The system's reliability was then tested and demonstrated in Chapter 6.

The results show that the R3TOS can keep the system fault-free at all times by rapidly detecting and fixing transient faults, and conveniently reallocating tasks onto non-damaged logic resources. Thereby, the R3TOS permits a step up to the Safety Integrity Levels (SIL) required by current international safety criteria, such as IEC-61508. Moreover, the R3TOS successfully translates reliable and efficient electronic advances into system performance improvement, while also prolonging the lifetime of reconfigurable devices.

Multi-User Multi-Tasking Computing

Aiming at improving the efficiency of the usage of computing resources, multi-user, multi-tasking systems have become more popular since they allow for the sharing of computing resources among different users and running multiple tasks in parallel, whereby limited hardware resources can be more efficiently exploited. The R3TOS has moved this concept to a more advanced scenario, where not only can the same resource be shared for different purposes, but also the functionality of the hardware can be customised and specialised for a particular user or a specific application task.

The R3TOS is capable of creating a piece of hardware logic for a specific application task and allocating it to an optimal position at run-time, whereby both system performance and the efficiency of resource usage are improved. This was demonstrated in the context of a bioinformatics sequence alignment application in Chapter 7. The results show that the novel algorithm achieves ~25% increase in allocation efficiency, and overall system performance is improved by ~2.5x.

Energy Efficiency

With advances in technology, transistor density rapidly increases as predicted by Moore's law. As a result, high levels of heat have become a bottleneck in improving device performance, since the heat generated by the high clock frequency is too high for cooling to be effective [Patterson2008]. Hardware specialisation has alleviated this problem to some degree, by using dedicated hardware for a specific application, rather than using Von-Neumann architectures. However, such dedicated hardware also renders the logic resource; concerned too specialised to be used for other tasks. Consequently, since specialised hardware is not frequently used, idle resources will entail undesired power consumption, which severely reduces energy efficiency.

The R3TOS has successfully eliminated the energy consumed by idle hardware logic, by removing finished tasks from the chip or disabling the clock for a particular region. Thereby, the system can benefit from specialised hardware without introducing redundant power consumption. In addition, the R3TOS achieves low overheads in both configuration time and area, which further reduces unnecessary energy costs.

High Performance and Adaptive Computing

The R3TOS promotes the development of high performance and adaptive computing. Firstly, the R3TOS has significantly improved computing performance by means of hardware specialisation, such as highly pipelined PEs, rather than using traditional generic computing platforms such as General Purpose Processors (GPPs) and Application-Specific Instruction-set Processors (ASIPs). Moreover, in the R3TOS, a task is operated locally in an exclusive region, which allows the task to be clocked at a maximal frequency despite other tasks running. As a consequence, higher performance can be achieved by the hardware specialisation and operational dependency, which permits a step up to the level of server-based high performance computing. Last, but not least, the R3TOS is capable of autonomously adapting the computing workload in response to the currently available resources, promoting a means for future reconfigurable adaptive computing.

Apart from the aforementioned four types of impact, the R3TOS also promotes the real-time performance and high level programming features for reconfigurable operating systems. In real-time, the novel FAEDF algorithm has achieved up to 2x better efficiency in scheduling hardware tasks (see Chapter 4), providing a good QoS to meet real-time constraints especially in reconfigurable systems. On the other hand, the R3TOS supports the high-level programming of reconfigurable hardware, by giving hardware tasks a software look-and-feel, which facilitates its use by software-centric application developers.

7.4 Criticisms and Future Work

The main contribution of the R3TOS is that it proves the feasibility of realising a system model to allow hardware tasks to behave just like software tasks, whereby some direct benefits can be obtained such as reliability, higher performance and adaptability. However, the development of the R3TOS is still not fully accomplished due to the limitations of devices and technologies. The following sections firstly give some critiques of the developments so far, and then propose future work related to this research.

Criticisms

The main limitation of the R3TOS system is configuration speed. Although the ICAP driver has achieved the maximal throughput of 400MB/s, the time overhead for configuring a task is still measured in the range of tens of microseconds. In effect, this bottleneck is caused by the limited bandwidth of the configuration memory, which is similar to the memory bottleneck in the Von Neumann architecture. To circumvent this bottleneck, the compressed bitstream is used to replicate multiple frames using MFW, which increases the speed by ~20x. Nevertheless, the time overhead is still too large to support frequent task swapping, such as the context switching used in preemptive scheduling algorithms. Besides above, the configuration speed limits the throughput of ICAP-based communication, making it feasible only for Low-bandwidth communication tasks. In order to solve this bottleneck, more advanced memory technologies need to be applied to the FPGA configuration memory, such as DDR4 memory which has a throughput of 21GB/s [Shilov2012].

In addition, the R3TOS is currently highly specified for particular chips and applications, making it difficult to be ported to other devices for different applications. One reason for this is that the implementation of the R3TOS relies on a set of specific low-level constraints, which have to be manually placed since they cannot be automatically managed by generic synthesis tools. For example the system's static region has to cover all of the essential elements, such as the ICAP, ECC, and IO ports, which are different in various chips. As a result, considerable effort will be required in updating the R3TOS to a different chip or board. Besides, the increasing heterogeneity of reconfigurable devices makes it difficult to area-constrain a task at low-level, as well as flexibly placing tasks to arbitrary positions. Consequently, only a limited number of application tasks requiring homogenous resources can benefit most from the system's capabilities.

Moreover, the reliability of the system is tested by injecting faulty bits into the configuration memory using the ICAP, which is only an emulation of a radiative environment. In the light of this, a real fault tolerance test needs to be carried out by

physically exposing devices to real radiation, rather than using artificial on-chip mechanisms.

Future work

First of all, a realistic solution has to be found to counter the configuration memory bottleneck. Besides the fact that forthcoming advances in technology, such as higher bandwidth memories may become available, the Network on Chips (NoCs) can be used to release the workload of the reconfigurable port, allowing the ICAP to be dedicated for use in configuring tasks only. For example, the NoC can provide bi-directional high bandwidth channels between the main CPU and tasks, which can replace the ICAP polling mechanism. However, NoCs may introduce extra footprints fragmenting the reconfigurable region and reducing allocation flexibility. Therefore, a balance-point needs to be found, which optimises the overall performance while trading off configuration speed and allocation flexibility.

Since the current R3TOS is highly specified to particular devices and applications, more universal solutions should be investigated to support for more standardised programming interfaces for both system and application development. In the first case, to standardise system development, the R3TOS has to become more compatible with commercial synthesis tools such as Xilinx XST. For instance, the low-level constraints of the R3TOS can be encapsulated into one single script file, which can be automatically recognised and translated by the hardware synthesiser during system implementation. Secondly, the development of user applications should be carried out in line with state-of-the-art parallel computing platforms and APIs such as NVIDIA CUDA and OpenCL, which can provide extra support for high-level dependency analysis and logic partitioning. Thus, the future development of the R3TOS as well as its applications should move toward being more efficient, efficacious and intelligent, by using other standard high-level synthesis tools rather than relying on inefficient manual work.

Apart from the previously mentioned works, the development of DPR techniques needs a key application to emphasize its fundamental importance and essentials,

where the reconfigurable system serves as the only solution, rather than a beneficial approach. Although the R3TOS has been demonstrated in the context of two applications, it can only be proven that the R3TOS improves system performance, rather than solving real problems. In addition, system performance should be tested and evaluated using a more standard benchmark, which should be built upon classical DPR-based applications. In the light of the above, more representative applications should be addressed to promote DPR to become a more attractive technique used in industrial fields.

Apart from the aforementioned three long-term future research areas, some short-term work could be achieved. First of all, system integration is not yet fully complete and is still on going at the time of writing this thesis. The fully integrated system will have all functionalities available at the same time and a more easy-to-use interface will be available to application developers.

While integrating the system, efforts have to be made to cope with resource heterogeneity, which is increasing with the appearance of more advanced devices. For example, in the latest FPGAs, not only are the traditional resources such as CLBs, BRAMs, DSPs and IOBs available, but also new specific blocks such as EMACs are integrated. In the light of this, the task allocator should be enhanced to deal with more diverse resource, as well as the low-level interconnections of hardware resources. One of the difficulties here is the lack of configuration information released by manufacturers, such as bitstream mapping and SB connections, which are currently obtained by means of reverse engineering carried out by researchers, resulting in a longer development time.

One interesting area of research would be to develop more intelligent allocation mechanisms. For example, a temperature sensor can be used to keep track of the temporal heat on the chip, whereby the task in a hot region can be reallocated to a cool area to avoid overloading the resources [Lopez-Buedo2004]. Another idea is to make the R3TOS system kernel (static region) reallocatable, and thereby fault-tolerance would be further improved since the system could cope with damaged resource occurring in the static region. However, system reallocation involves a

number of static primitives, such as the ICAP, which increases the level of difficulty of such work.

Last, but not least, the R3TOS is being updated by present research group members to more advanced FPGA chips, including a Xilinx Virtex-6 FPGA (XC6VLX240T) on a ML605 board, and a Xilinx Zynq FPGA (Zynq-7000) on a ZedBoard [Xilinx2013a, Xilinx2013b]. The new generation of chips provides the R3TOS with more powerful device support. For example, The Virtex-6 FPGA provides a larger chip area (37680 slices and 416 BRAMs) with higher processing (40 nm), giving better allocation capability and flexibility. The Zynq FPGA has two embedded ARM Cortex-A9 processors, which are ideal for implementing the R3TOS's main CPU. In particular, the configuration port in the Zynq FPGA is significantly improved. It is reported in the latest Zynq user guide that a new generation of the ICAP, called the Processor Configuration Access Port (PCAP), supports up to 200MHz clock frequency, which doubles the maximum throughput of the ICAP from 400MB/s to 800MB/s [Xilinx2013a]. The evolution of technology also implies that industrial vendors are paying more attention to DPR–based techniques. In the light of this, the R3TOS should be updated with the latest products to allow it to maximally benefit from the most advanced DPR technologies.

References

[Ahmadinia2004a] A. Ahmadinia, C. Bobda, D. Koch, M. Majer and J. Teich, "Task scheduling for heterogeneous reconfigurable computers", In *Proceedings of the International Symposium on Integrated Circuits and System Design,* pp. 22–27, 2004

[Ahmadinia2004b] A. Ahmadinia, C. Bobda, M. Bednara and J. Teich, "A new approach for on-line placement on reconfigurable devices", In *Proceedings of the International Parallel and Distributed Processing Symposium,* 2004

[Ahmadinia2005] A. Ahmadinia, C. Bobda, J. Ding, M. Majer, J. Teich, S. Fekete and J. van der Veen, "A practical approach for circuit routing on dynamic reconfigurable devices", In *Proceedings of the IEEE International Workshop on Rapid System Prototyping,* pp.84–90, 2005

[Ahmadinia2007] A. Ahmadinia, C. Bobda, S. P. Fekete, J. Teich and J. van der Veen, "Optimal free-space management and routing-conscious dynamic placement for reconfigurable devices", *IEEE Transactions on Computers,* 56(5):673–680, 2007

[Al Farisi2011] B. Al Farisi, K. Heyse, K. Bruneel, and D. Stroobandt, "Memory-efficient and fast run-time reconfiguration of regularly structured designs", In *Proceedings of the International Conference on Field-Programmable Logic and Applications,* 2011

[Altera2010] Altera Inc., "*Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs (WP-01137-1.0)*", Technical report, 2010

[Andrews2005] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, M and R. Sass, "hthreads: A hardware/software co-designed multithreaded RTOS kernel", In *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation,* 2005

[Angermeier2011] J. Angermeier, D. Ziener, M. Glass and J. Teich, "Stress-aware module placement on reconfigurable devices", *International Conference on Field-Programmable Logic and Applications,* pp.277–281, 2011

[Backus1978] J. Backus, "Can programming be liberated from the von Neumann style: a functional style and its algebra of programs" *Communications of the ACM,* vol 21, pp. 613–641, 1978

[Banerjee2006] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "PARLGRAN: parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures". In *Proceedings of the Asia and South Pacific Design Automation Conference,* pp. 491–496, 2006

[Bazargan2000] K.Bazargan, R.Kastner, M.Sarrafzadeh, "Fast template placement for reconfigurable computing systems", In *Design & Test of Computers.* vol. 17, pp. 68–83, 2000

[Becker2007] T. Becker, W. Luk, and Peter Y. K. Cheung, "Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration", *IEEE Symposium on Field-Programmable Custom Computing Machines,* pp. 35-44, 2007.

[Becker2010] T. Becker, M. Koester and W. Luk, "Automated placement of reconfigurable regions for relocatable modules", In *Proceedings of IEEE International Symposium on Circuits and Systems,* pp. 3341 –3344, 2010

[Bellato2004] M. Bellato, P. Bernardi, D. Bortolato, A. Candelori, M. Ceschia, A. Paccagnella, M. Rebaudengo, M. Sonza Reorda, M. Violante and P. Zambolin, "Evaluating the effects of SEUs affecting the configuration memory of an SRAM-based FPGA", In *Proceedings of the Conference on Design, Automation and Test in Europe,* 2004

[Benkrid2000] K. Benkrid, D. Crookes, J. Smith and A. Benkrid, "High Level Programming for Real Time FPGA Based Video Programming", In *Proceedings of the IEEE International Conference on Acoustic, Speech and Signal Processing, ICASSP'2000, Istanbul, Volume VI,* pp. 3227-3231, 2000

[Benkrid2004] K. Benkrid, D. Crookes, “From application descriptions to hardware in seconds: a logic-based approach to bridging the gap”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, TVLSI 12*, vol. 4, pp. 420–436, 2004

[Benkrid2008] K.Benkrid, “High Performance Reconfigurable Computing: From Applications to Hardware” *IAENG International Journal of Computer Science,* vol. 35, no. 1, 2008

[Benkrid2009]K. Benkrid, Y. Liu, A. S. Benkrid. “A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment”. *IEEE Transactions. VLSI Systems,* Vol. 17, no. 4, pp. 561–70, 2009

[Benkrid2012]K. Benkrid, A. Akoglu, C. Ling, Y. Song, Y. Liu and X. Tian, “High Performance Biological Pairwise Sequence Alignment: FPGA vs. GPU vs. Cell BE vs. GPP”, *International Journal of Reconfigurable Computing,* April 2012, (Accepted for publication)

[Bertels2011] K. Bertels, “*Hardware/software co-design for heterogeneous multi-core platforms: The Hartes toolchain*”, Springer, 2011

[Blodget2003]B. Blodget, S. McMillan, and P. Lysaght, “A lightweight approach for embedded reconfiguration of FPGAs,” in *Proc. Design, Automation and Test in Europe Conference and Exhibition,* pp. 399–400, 2003

[Brebner1996] G. J. Brebner, “A virtual hardware operating system for the Xilinx XC6200”, In *Proceedings of the International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pp. 327–336, 1996

[Brennan2005] D. Brennan et al., “Application of DNA microarray technology in determining breast cancer prognosis and therapeutic response,” *J. Expret Opin. Biol. Ther.*, vol. 5, no. 8, pp. 1069–1083, Aug. 2005.

[Buttazzo2004] G. C. Buttazzo, “Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications (Real-Time Systems Series)”, *Springer-Verlag TELOS,* Santa Clara, CA, USA, 2004

[Carver2008] J. Carver, N. Pittman and A. Forin, "*Relocation and automatic floor-planning of FPGA partial configuration bitstreams*", Technical report, Microsoft Research. 2008

[Chandra2000] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, "*Parallel programming in OpenMP*", Morgan Kaufmann Publisher, 2000

[Churcher1995] S. Churcher, T. Kean and B. Wilkie, "The XC6200 FastMap™ processor interface," In *Field-Programmable Logic and Applications, Springer Publishing Company*, pp. 36--43, 1995

[Compton2002] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software". *ACM Computing Survey*, 34:171–210, 2002

[Corbett2012] J. D. Corbett, "*The Xilinx isolation design flow for fault-tolerant systems (WP412)*", Technical report, Xilinx Inc, 2012

[Couch2011] J. Couch and P. Athanas, "An analysis of implanted antennas in Xilinx FPGAs", In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, pp. 1–6. 2011

[Coussy2008] P. Coussy and A. Morawiec, "*High-Level Synthesis: from Algorithm to Digital Circuit*", Springer Publishing Company, Incorporated, 1st edition, 2008

[Cui2007] J. Cui, Z. Gu, W. Liu and Q. Deng, "An efficient algorithm for online soft real-time task placement on reconfigurable hardware devices", In *Proceedings of the IEEE International Symposium on Object and Component-Oriented Real- Time Distributed Computing*, pp. 321–328, 2007

[Danne2005] K. Danne and M. Platzner, "Periodic real-time scheduling for FPGA computers", In *Proceedings of the International Workshop on Intelligent Solutions in Embedded Systems*, pp. 117–127, 2005

[Dietterich1995] T. G. Dietterich and G. Bakiri, "*Solving multiclass learning problems via error-correcting output codes*", arXiv preprint cs/9501101, 1995

[Dittmann2007] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling", In *Proceedings of the Conference on Design, Automation and Test in Europe,* pages 123–128, 2007

[Donato2005] A. Donato, F. Ferrandi, M. Redaelli, M. D. Santambrogio and D. Sciuto, "Caronte: a complete methodology for the implementation of partially dynamically self-reconfiguring systems on FPGA platforms," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM),* pp. 321—322, 2005

[Dubach2010] C. Dubach, T. M. Jones, E. Bonilla, and M. F. P. O'Boyle, "A predictive model for dynamic micro architectural adaptivity control". In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture",* 2010

[Durbin1998] R. Durbin, S. Eddy, A. Krogh and G. Mitchison, "*Biological sequence analysis: probabilistic models of proteins and nucleic acids*" Cambridge University Press, Cambridge UK; 1998.

[Dye2010] D. Dye, "*Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite*", Xilinx white paper, May 30, 2012

[Ebrahim2012] A. Ebrahim, K. Benkrid, X. Iturbe, and C. Hong, "A Novel High-Performance Fault-Tolerant ICAP Controller", *NASA/ESA Conference on Adaptive Hardware and Systems,* 2012, Erlangen, Germany. (Accepted for publication)

[Ejnioui2005] A. Ejnioui and R. F. DeMara, "Area reclamation strategies and metrics for SRAM-based reconfigurable devices", In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms,* pp. 196–202, 2005

[El Farag2007] A. El Farag, H. El-Boghdadi and S. Shaheen, "Improving utilization of reconfigurable resources using two-dimensional compaction" *Journal of Supercomputing,* 42:235–250, 2007

[Feng2010] S. Feng, S. Gupta, A. Ansari and S. A. Mahlke, “Maestro: Orchestrating lifetime reliability in chipmultiprocessors” In *Proceedings of the International Conference on High-Performance Embedded Architectures and Compilers,* pp. 186–200, 2010

[Fuller2000] E. Fuller, M. Caffrey, A. Salazar, C. Carmichael, J. Fabula, “ Radiation testing update, SEU mitigation, and availability analysis of the Virtex FPGA for space reconfigurable computing,“ In *Proceedings of Conference on. IEEE Nuclear and Space Radiation Effects,* pp. 31–41, 2000.

[Gohringer2010a] D. Gohringer, M. Hubner, E. N. Zeutebouo and J. Becker, “Operating system for runtime reconfigurable multiprocessor systems”, *International Journal of Reconfigurable Computing,* 2010

[Gohringer2010b] D. Gohringer, M. Hubner, L. Hugot-Derville and J. Becker, “Message passing interface support for the runtime adaptive multi-processor system-on-chip RAMPSoC”, In *Proceedings of the IEEE International Conference IC-SAMOS,* pp. 357– 364, 2010

[Gohringer2011] G. Gohringer, M. Hubner, E. N. Zeutebouo and J. Becker, “Operating system for runtime reconfigurable multiprocessor systems”, *International Journal of Reconfigurable Computing,* vol.2011, pp.3, 2011

[Golub1999] T. R. Golub et al., “Molecular classification of cancer: class discovery and class prediction by gene expression monitoring,” *Science J.,* vol. 286, no. 5439, pp. 531–537, Oct. 1999.

[Gotoh1982] O. Gotoh. “An improved algorithm for matching biological sequences”, *Journal of Molecular Biology,* vol. 162, no. 3, pp. 705, 1982

[Hamming1950] R. W. Hamming, “Error detecting and error correcting codes”, *Bell System technical journal,* vol. 29, no. 2, pp. 147– 160, 1950

[Handa2004] M. Handa and R. Vemuri, "An efficient algorithm for finding empty space for online FPGA placement", In *Proceedings of the Annual Design Automation Conference,* pp. 960–965, 2004

[Heiner2008] J. Heiner, N. Collins, and M. Wirthlin, "Fault tolerant ICAP controller for high-reliable internal scrubbing," In *Proc. Aerospace Conference,* pp. 1– 10, 2008.

[Hong2011a] C. Hong, K. Benkrid, X. Iturbe, A. T. Erdogan and T. Arslan, "An FPGA Task Allocator with Preliminary First-Fit 2D Packing Algorithms", *NASA/ESA Conference on Adaptive Hardware and Systems (AHS),* pp. 264– 270, 2011, California, USA.

[Hong2011b] C. Hong, K. Benkrid, X. Iturbe, A. Ebrahim, and T. Arslan, "Efficient On-Chip Task Scheduler and Allocator for Reconfigurable Operating Systems" *IEEE Embedded Systems Letters,* vol. 3, issue. 3, pp. 85– 88, 2011.

[Hong2012a] C. Hong, K. Benkrid, X. Iturbe and A. Ebrahim, "Design and Implementation of Fault-tolerant Soft Processors on FPGAs", *International Conference on Field-Programmable Logic and Applications (FPL),* pp. 683–686, 2012, Oslo, Norway.

[Hong2012b] C. Hong, K. Benkrid, A. Ebrahim and X. Iturbe, "Virtual Shared Memory Architecture for Inter-Task Communication in Partial Reconfigurable Systems", *International Conference on Microelectronics (ICM),* pp.1–4, 2012, Algiers, Algeria.

[Hong2013a] C. Hong, K. Benkrid, X. Iturbe and H. Hussain, "Efficient Run-time System Support for High Performance Reliable Reconfigurable Systems", *Journal of Computational Intelligence and Electronic Systems (JCIES)* (Accepted for publication)

[Hong2013b] C. Hong, K. Benkrid, N. Isa and X. Iturbe, "A Run-time Reconfigurable System for Adaptive High Performance Efficient Computing", *ACM SIGARCH Computer Architecture News,* 2013 (Accepted for publication)

[Horta2002] E.L.Horta, J.W.Lockwood, D.E.Taylor, and D.Parlour, "Dynamic hardware plugins in an FPGA with partial run-time reconfiguration," In *Proceeding of 39th annual Design Automation Conference,* pp. 343– 348, 2002

[Hubner2006] M. Hubner, C. Schuck and J. Becker, "Elementary block based 2-dimensional dynamic and partial reconfiguration for Virtex-II FPGAs", In *Proceedings of the International Conference on Parallel and Distributed Processing,* pp. 196–196, 2006

[Hussain2012a] H. Hussain, K. Benkrid, C. Hong, and H. Seker, "An Adaptive FPGA Implementation of Multi-Core K-Nearest Neighbour Ensemble Classifier using Dynamic Partial Reconfiguration," *International Conference on Field-Programmable Logic and Applications (FPL),* pp. 627– 630, 2012, Oslo, Norway

[Hussain2012b] H. Hussain, "*Dynamically and Partially Reconfigurable Hardware Architectures for High Performance Microarray Bioinformatics Data Analysis*", PhD thesis, University of Edinburgh, UK, 2012

[Hutchings1999] B. Hutchings, "A CAD suite for high-performance FPGA design", In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'99,* pp. 12–4, 1999

[Inam2011] R. Inam, J. Maki-Turja, M. Sjodin, S. Ashjaei, H. Mohammad and S. Afshar, "Support for hierarchical scheduling in FreeRTOS", *Conference on Emerging Technologies \& Factory Automation (ETFA),* pp. 1–10, 2011

[Isa2012] M.N.Isa, K.Benkrid and T.Clayton "Efficient Architecture and Scheduling Technique for Pairwise Sequence Alignment" ACM, *SIGARCH Computer Architecture News,* 2012, (Accepted for publication)

[Isa2013] M.N.Isa, "*High Performance Reconfigurable Architectures for Biological Sequence Alignment*", PhD thesis, University of Edinburgh, UK, 2013

[Ismail2011] A. Ismail and L. Shannon, "FUSE: Front-end user framework for O/S abstraction of hardware accelerators", In *Proceedings of the Annual IEEE*

International Symposium on Field-Programmable Custom Computing Machines, pp. 170–177, 2011

[Iturbe2009] X. Iturbe, M. Azkarate, I. Martinez, J. Perez, and A. Astarloa, "A novel SEU, MBU and SHE handling strategy for Xilinx Virtex-4 FPGAs", In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pp. 569–573, 2009

[Iturbe2010a] X. Iturbe, K. Benkrid, T. Arslan, I. Martinez, M. Azkarate and M. D. Santambrogio, "A Roadmap for Autonomous Fault-Tolerant Systems", In *Proceedings of the International Conference on DASIP*, pp. 311-321, 2010.

[Iturbe2010b] X.Iturbe, K.Benkrid, A.T.Erdogan, T.Arslan, M.Azkarate, I.Martinez, and A.Perez, "R3TOS: A reliable reconfigurable real-time operating system," In *Proceedings of the International Conference on AHS*, pp. 99-104, 2010

[Iturbe2010c] X. Iturbe, K. Benkrid, T. Arslan, I. Martinez, M. Azkarate and A. Morales-Reyes, "Evolutionary dynamic allocation of relocatable modules onto partially damaged Xilinx FPGAs", In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 211–217, 2010

[Iturbe2011a] X.Iturbe, K.Benkrid, A.Ebrahim, C.Hong, T. Arslan, and I.Martinez, "Snake: An Efficient Strategy for the Reuse of Circuitry and Partial Computation Results in High-Performance Reconfigurable Computing", In *Proceedings of the International Conference on ReConFig*, pp. 182-189, 2011

[Iturbe2011b] X. Iturbe, K. Benkrid, T. Arslan, R. Torrego, and I. Martinez, "Methods and Mechanisms for Hardware Multitasking: Executing and Synchronizing Fully Relocatable Hardware Tasks in Xilinx FPGAs ". *International Conference on Field-Programmable Logic and Applications (FPL'11)*, Crete, 2011.

[Iturbe2011c] X. Iturbe, K. Benkrid, T. Arslan, C. Hong, and I. Martinez, "Empty resource compaction algorithms for real-time hardware tasks placement on partially reconfigurable FPGAs subject to fault occurrence", In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, 2011

[Iturbe2011d] X. Iturbe, K. Benkrid, T. Arslan, C. Hong and I. Martinez, "Enabling FPGAs for future deep space exploration missions: Improving fault tolerance and computation density with R3TOS", In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, 2011

[Iturbe2012] X. Iturbe, K. Benkrid, R. Torrego, A. Ebrahim and T. Arslan, "Online clock routing in Xilinx FPGAs for high-performance and reliability", In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, 2012 (Accepted for publication)

[Iturbe2013a] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, T. Arslan, and I. Martinez, "Runtime scheduling, allocation and execution of real-time hardware tasks onto Xilinx FPGAs subject to fault occurrence", *International Journal of Reconfigurable Computing*, 2013 (Accepted for publication)

[Iturbe2013b] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, I. Martinez, T. Arslan and J. Perez, "R3TOS: A novel reliable reconfigurable real-time operating system for highly adaptive, efficient and dependable computing on FPGAs", *IEEE Transactions on Computers, Special Issue on "Adaptive Hardware and Systems",* 2013 (Accepted for publication)

[Iturbe2013c] X. Iturbe, "*Design and Implementation of a Reliable Reconfigurable Real-Time Operating System*", PhD thesis, University of Edinburgh, UK, 2013

[Jara-Berrocal2010] A. Jara-Berrocal and A. Gordon-Ross, "VAPRES: A virtual architecture for partially reconfigurable embedded systems", In *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 837– 842, 2010

[Jovanovic2007] S. Jovanovic, C. Tanougast and S. Weber, "A hardware preemptive multitasking mechanism based on scan-path register structure for FPGA-based reconfigurable systems", In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 358–364, 2007

[Jozwik2010] K. Jozwik, H. Tomiyama, S. Honda and H. Takada, "A novel mechanism for effective hardware task preemption in dynamically reconfigurable

systems" In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pp. 352–355, 2010

[Kalte2005] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems", In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pp. 223–228, 2005

[Kalte2005] H. Kalte and M. Porrmann, "Context Saving and Restoring for Multitasking in Reconfigurable Systems", In *Proc. of the International Conference on Field-Programmable Logic and Applications*, pp. 223–228, 2005.

[Katz2003] D.S.Katz, and R.R.Some, "NASA advances robotic space exploration," in *Computer*, vol. 36, issue. 1, pp. 52–61, 2003.

[Koch2008] D. Koch, C. Beckhoff and J. Teich, "ReCoBus builder – a novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs", In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pp. 119–124, 2008

[Koch2009] D. Koch, "*Architectures, Methods, and Tools for Distributed Run-time Reconfigurable FPGA-based Systems*", PhD thesis, University of Erlangen-Nuremberg, Germany, 2009

[Koch2010a] D. Koch, C. Beckhoff and J. Torresen, "Obstacle-free two-dimensional online-routing for run-time reconfigurable FPGA-based systems", In *Proceedings of the International Conference on Field-Programmable Technology*, pp. 208–215, 2010

[Koch2010b] D. Koch, C. Beckhoff and J. Torresen, "Zero logic overhead integration of partially reconfigurable modules", In *Proceedings of the Symposium on Integrated Circuits and System Design*, pp. 103–108, 2011

[Koester2009] M. Koester, W. Luk, J. Hagemeyer and M. Porrmann, "Design optimizations to improve placeability of partial reconfiguration modules", In

Proceedings of the Conference on Design, Automation and Test in Europe, pp. 976–981, 2009

[Kopetz2011] H. Kopetz, "*Real-Time Systems: Design Principles for Distributed Embedded Applications*", Springer-Verlag, 2nd edition, 2011

[Korf2011] S. Korf, D. Cozzi, M. Koester, J. Hagemeyer, M. Porrmann, U. Ruckert and M. Santambrogio, "Automatic HDL-based generation of homogeneous hard macros for FPGAs", In *Proceedings of the IEEE Annual International Symposiumon Field-Programmable Custom ComputingMachines,* pp. 208–215, 2011

[Kuon2008] I.Kuon, R.Tessier, and J.Rose, "FPGA architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation,* vol. 2, pp. 135–253, 2008

[Lala1994] J. Lala and R. Harper, "Architectural principles for safety-critical real-time applications", *Proceedings of the IEEE,* 82(1):25–40, 1994

[LeCao2009] K. Le Cao and G. McLachlan, "Statistical Analysis on Microarray Data: Selection of Gene Prognosis Signatures," In *Computational Biolog: Issues and Application in Oncology (Applies Bioinformatics and Biostatistics in Cancer Research series), T. Pham (Ed.),* 1st ed., New York : Springer, 2009, ch. 3, pp. 55–75, 2009

[Lesea2005] A.Lesea, S.Drimer, J.J.Fabula, C.Carmichael, and P.Alfke, "The Rosetta experiment: Atmospheric soft error rate testing in differing technology FPGAs," *IEEE Transactions on Device and Materials Reliability,* vol. 5, issue. 1, pp. 317–328, 2005.

[Leung1982] J. Y. T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks" *Performance Evaluation,* 2(4):237–250, 1982

[Li2002] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation", In *Proceedings of*

the ACM/SIGDA International Symposium on Field-programmable Gate Arrays, pp. 187–195, 2002

[Liu1973] C. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment", *Journal of ACM*, 20(1):46 61, 1973

[Lopez-Buedo2004] S. Lopez-Buedo and E. Boemo, "Making visible the thermal behaviour of embedded microprocessors on FPGAs: a progress report", In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 79–86, 2004

[Lu2009] Y. Lu, T. Marconi, K. Bertels and G. Gaydadjiev, "Online task scheduling for the FPGA-based partially reconfigurable systems" In *Proceedings of the International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, pp. 216–230, 2009

[Lu2010] Y. Lu, T. Marconi, K. Bertels and G. Gaydadjiev, "A communication aware online task scheduling algorithm for FPGA-based partially reconfigurable systems", In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom ComputingMachines*, pp. 65–68, 2010

[Lubbers2010] E. Lubbers, "*Multithreaded Programming and ExecutionModels for Reconfigurable Hardware*", PhD thesis, University of Paderborn, Germany, 2010

[Macgrefor2002] P. Macgrefor and J. Squire, "Application of Microarray to the Analysis of Gene Expression in Cancer," *J. Clinical Chemistry, v*ol. 48, no. 8, pp. 1170–1177, Aug. 2002.

[Manolakos2010] E. Manolakos and I. Stamoulias,"IP-cores design for the kNN classifier," In *Proceedings IEEE International. Symposium, Circuits and Systems*, pp. 4133–4136, May 2010

[Marconi2010] T. Marconi, Y. Lu, K. Bertels and G. Gaydadjiev, "3D compaction: A novel blocking-aware algorithm for online hardware task scheduling and

placement on 2D partially reconfigurable devices" In *Proceedings of the International Symposium on Appied Reconfigurable Computing*, pp. 194–206, 2010

[Mignolet2003] J. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde and R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable System-on-Chip", In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2003

[Morandi2008] M. Morandi, M. Novati, M. D. Santambrogio and D. Sciuto, "Core allocation and relocation management for a self dynamically reconfigurable architecture", In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pp. 286–291, 2008

[Oliver2005] T. F. Oliver, B. Schmidt, D. L. Maskell "Reconfigurable architectures for bio-sequence database scanning on FPGAs". *IEEE Transactions. Circuits and Systems II*, vol. 52, no.12, pp. 851-855, 2005.

[Osamu1982] G. Osamu, "An improved algorithm for matching biological sequences", *Journal of Molecular Biology*, 1982;162(3):705-8.

[Palesi2007] M. Palesi, S. Kumar, R. Holsmark and V. Catania, "Exploiting communication concurrency for efficient deadlock free routing in reconfigurable NoC platforms", In *Proceedings of the IEEE International Parallel andDistributed Processing Symposium*, pp. 1–8, 2007

[Patterson2008] D. A. Patterson, and J. L. Hennessy, "Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)", *Morgan Kaufmann Publishers Inc.*, San Francisco, CA, USA, 4th edition, 2008

[Raabe2008] A. Raabe, P. A. Hartmann and J. K. Anlauf, "ReChannel: Describing and simulating reconfigurable hardware in systemC", *ACM Transactions on Design Automation of Electronic Systems*, 13:1–18, 2008

[Redaelli2009] F. Redaelli, M. D. Santambrogio and S. O. Memik, "An ILP formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures" *International Journal of Reconfigurable Computing,* 1–12, 2009

[Sander2008] O. Sander, L. Braun, M. Hubner and J. Becker, "Data reallocation by exploiting FPGA configuration mechanisms", In *Proceedings of the International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, pp. 312–317, 2008

[Schumacher2012] P. Schumacher, "*SEU emulation environment (WP414)*", Technical report, Xilinx Inc, 2012

[Sedcole2005] P. Sedcole, B. Blodget, J. Anderson, P. Lysaght, T. Becker, "Modular partial reconfiguration in Virtex FPGAs," in *Proceedings of International Conference on Field-Programmable Logic and Applications*, pp. 211—216, 2005

[Sedcole2006] P. Sedcole, B. Blodget, T. Becker, J. Anderson and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs", *IEE Proceedings on Computers and Digital Techniques,* 153(3):157–164, 2006

[Sedcole2007] P. Sedcole, P. Y. K. Cheung, G. A. Constantinides and W. Luk, "Run-time integration of reconfigurable video processing systems", *IEEE Transactions on Very Large Scale Integration Systems,* 15:1003–1016, 2007

[Shayani2008] H. Shayani, P. Bentley and A. M. Tyrrell, "A cellular structure for online routing of digital spiking neuron axons and dendrites on FPGAs" In *Proceedings of the International Conference on Evolvable Systems: From Biology to Hardware*, pp. 273–284, 2008

[Shelburne2008] M. Shelburne, C. Patterson, P. Athanas, M. Jones, B. Martin and R. Fong, "Metawire: Using FPGA configuration circuitry to emulate a Network-on-Chip" In *International Conference on Field-Programmable Logic and Applications*, pp. 257–262, 2008

[Shilov2012] A. Shilov, "Leading memory makers show DDR4 prototypes at ISSCC", on line at http://www.xbitlabs.com/news/memory/display/20120224123543_Leading_Memory_Makers_Show_DDR4_Prototypes_at_ISSCC.html.

[Silva2010] M. L. Silva and J. C. Ferreira, "Creation of partial FPGA configurations at run-time", In *Proceedings of the Euromicro Conference on Digital System Design: Architectures,Methods and Tools,* pp. 80–87, 2010

[Simmler2000] H. Simmler, L. Levinson and R. Manner, "Multitasking on FPGA coprocessors", In *Proceedings of the International Workshop on Field-Programmable Logic and Applications,* pp. 121–130, 2000

[So2007] H. K. H. So, "*BORPH: An Operating System for FPGA-Based Reconfigurable Computers*", PhD thesis, University of California at Berkeley, USA, 2007

[So2008] H. K. H. So and R. Brodersen, "A unified hardware/ software runtime environment for FPGA-based reconfigurable computers using BORPH", *ACM Transactions on Embedded Computing Systems,* 7(2):1–28, 2008

[Sohanghpurwala2011] A. Sohanghpurwala, P. Athanas, T. Frangieh and A. Wood, "OpenPR: An open-source partial-reconfiguration toolkit for Xilinx FPGAs", In *Proceedings of the IEEE International Symposium on Parallel and Distributed ProcessingWorkshops and Phd Forum,* pp. 228–235, 2011

[Srinivasan2006] S. Srinivasan, P. Mangalagiri, Y. Xie, N. Vijaykrishnan and K. Sarpatwari, "FLAW: FPGA lifetime awareness" In *Proceedings of the Annual Design Automation Conference,* pp. 630–635, 2006

[Steiger2003] C. Steiger, H. Walder and M. Platzner, "Heuristics for online scheduling real-time tasks to partially reconfigurable devices", In *Proceedings of the International Conference on Field-Programmable Logic and Applications,* pp. 575–584, 2003

[Stensgaard2008] M. Stensgaard and J. Sparso, "ReNoC: A Network-on-Chip architecture with reconfigurable topology" In *ACM/IEEE International Symposium on Networks-on-Chip,* pp. 55–64, 2008

[Sundarajan2010] P. Sundarajan, "High-performance computing using FPGAs (WP375)". *Technical report, Xilinx Inc.* 2010

[Suris2008] J. Suris, C. Patterson and P. Athanas, "An efficient runtime router for connecting modules in FPGAs", In *International Conference on Field-Programmable Logic and Applications,* pp. 125–130, 2008

[Suris2008] J. Suris, M. Shelburne, C. Patterson, P. Athanas, J. Bowen, T. Dunham and J. Rice, "Untethered on-the-fly radio assembly with Wires-On-Demand," In *IEEE National Aerospace and Electronics Conference,* pp. 229 233, 2008

[Tabero2004] J.Tabero, J.Septi´en, Mecha.H, Mozos.D, "A low fragmentation heuristic for task placement in 2D RTR HW management," In *Proceeding of International Conference on Field-Programmable Logic and Applications,* pp. 241—250, 2004.

[Tabero2006] J. Tabero, J. Septien, H. Mecha and D. Mozos, "Task placement heuristic based on 3D-adjacency and look-ahead in reconfigurable systems", In *Proceedings of the Asia and South PacificDesign Automation Conference,* pp. 396–401, 2006

[Tahir2006] M. A. Tahir and J. Smith, "Improving Nearest Neighbor Classifier using Tabu Search and Ensemble Distance Metrics," In *Proceeding of IEEE International Conference on Data Mining,* pp. 1086–109, 2006

[Thijs2007] G. Thijs, "Gene Regulation Bioinformatics of Microarray Data," In *Genomics and Proteomics Engineering in Medicine and Biology (IEEE Press Series in Biomedical Engineering), M. Akay (Ed.), Canada: John Wiley & Sons,* 2007, ch. 3, pp. 55–128.

[Thomas2002] D. E. Thomas and P. R. Moorby, "*The Verilog® Hardware Description Language*", vol.2, Springer press, 2002

[Tian2010] X. Tian and K. Benkrid, "High-performance quasi monte carlo financial simulation: FPGA vs. GPP vs. GPU" *ACM Transactions on Reconfigurable Technology and Systems,* 3(4):1–22, 2010

[Todman2005] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk and P. Cheung, "Reconfigurable computing: architectures and design methods". *IEE Proceedings on Computers and Digital Techniques,* 152(2):193–207, 2005

[Torrego2013] R. Torrego, I. Val, E. Muxika, X. Iturbe and K. Benkrid, "Data Coding Functions for Software Defined Radios implemented on R3TOS", *International Conference on Field-Programmable Logic and Applications (FPL),* 2012, Oslo, Norway.

[Tse2012] H. T. Tse, "*Accelerating Reconfigurable Financial Computing*" PhD thesis, Imperial College London, UK, 2012

[Upegui2006] A. Upegui and E. Sanchez, "Evolving hardware with self-reconfigurable connectivity in Xilinx FPGAs", In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems,* pp. 153–162, 2006

[van der Veen2005] J. van der Veen, S. P. Fekete, M. Majer, A. Ahmadinia, C. Bobda, F. Hannig and J. Teich, "Defragmenting the module layout of a partially reconfigurable device" In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms,* pp. 92–104, 2005

[Walder2005] H. Walder, "*Operating SystemDesign for Partially Reconfigurable Logic Devices*" PhD thesis, Swiss Federal Institute of Technology Zurich, Switzerland, 2005

[Wigley2001] G. Wigley and D. Kearney, "The first real operating system for reconfigurable computers", In *Proceedings of the Australasian Conference on Computer Systems Architecture,* pp. 130–137, 2001

[Wigley2001] G. Wigley and D. Kearney, "The first real operating system for reconfigurable computers", In *Proceedings of the Australasian Conference on Computer Systems Architecture,* pp. 130–137, 2001

[Wigley2002] G. Wigley and D. Kearney, "Research issues in operating systems for reconfigurable computing", In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms,* 2002

[Xilinx1997] Xilinx Inc., "*XC6200 field programmable gate arrays. Technical report*", 1997

[Xilinx1999] Xilinx Inc., "*FPGA Editor Guide*," *2.1i, 1999*

[Xilinx2001] Xilinx Application Note, "*Triple Module Redundancy Design Techniques for Virtex FPGAs*," XAPP197, 2001

[Xilinx2002] Xilinx Inc., "*MicroBlaze™ SoftwareReferenceGuide*", 2002

[Xilinx2007] Xilinx Inc., "*Virtex-II Pro and Virtex-II Pro X FPGA*", UG012, 2007

[**Xilinx2008**] Xilinx Inc., "*Virtex-4 FPGA User Guide*," UG070, 2008

[Xilinx2009] Xilinx Inc., "*Xilinx Virtex-4 FPGA Configuration Guid*e," UG071, June 9, 2009

[Xilinx2009b] Xilinx Inc., "*Data2MEM User Guide*," UG658, June 24, 2009

[Xilinx2010a] Xilinx Inc., "*Xilinx Partial Reconfiguration User Guide*," UG702, October, 2010

[Xilinx2010b] Xilinx Inc., "*LogiCORE IP XPS HWICAP*," DS586 July 23, 2010

[Xilinx2011a] Xilinx Inc., "*Device Reliability Report*," UG116, 2011

[Xilinx2011b] Xilinx Inc., "*PicoBlaze 8-bit Embedded Microcontroller User Guide*," UG129, 2011

[Xilinx2012a] Xilinx Inc., "*Xilinx Virtex-5 FPGA Configuration Guide*," UG191, October 19, 2012

[Xilinx2012b] Xilinx Inc., "*Partial Reconfiguration Tutorial*," UG743, May 08, 2012

[Xilinx2012c] Xilinx Inc., "*Virtex-5 FPGA User Guide*," UG190, March 16, 2012

[Xilinx2013a] Xiilnx Inc., "*Zynq-7000 All Programmable SoC Technical Reference Manual*" UG585, March 7, 2013

[Xilinx2013b] Xilinx Inc., "*Virtex-6 FPGA Configuration User Guide*," UG360, April 18, 2013

[Yamauchi2000] A. M. Yamauchi, "*Error detection and correction code for data and check code fields,*" US Patent 6,041,430, Google Patents, 2000

[Zhang2006] Y. Zhang, J. Roivainen and A. Mammela, "Clock-gating in FPGAs: A novel and comparative evaluation" In *Proceedings of the EUROMICRO Conference on Digital SystemDesign: Architectures,Methods and Tools,* pp. 584–590, 2006

[Zhou2005] B. Zhou, W. Qiu,W and C. Peng, "An operating system framework for reconfigurable systems", In *Proceedings of the The International Conference on Computer and Information Technology,* pp. 788–792, 2005

Appendix

Appendix-1

Operation Timing of Fault Tolerant Microprocessor

This appendix illustrates the detailed timing information of the fault tolerant microprocessor. The implemented operations include non-branching operation, JUMP, CALL, RETURN, INTERRUPT and RETURNI operations.

Non-branching operation

When no branching operation is executed, the fetching, decoding, and executing of instructions can be pipelined continuously, which is similar to the standard operation with the only difference being that instructions are executed with a latency of three clock cycles, rather than one cycle. In such a situation, there is no extra change needed to manipulate instructions. For example, in **Error! Reference source not found.**, the PicoBlaze PC is incremented sequentially from 000 to 003 every two cycles. The ECC-BRAM gives two instructions every four cycles, since it omits the lowest bit of PC, and stores two instructions in one memory address. The two instructions become available two cycles after the new address is given, and one more cycle is used by EPA multiplexing, before they are finally executed by the PicoBlaze core.

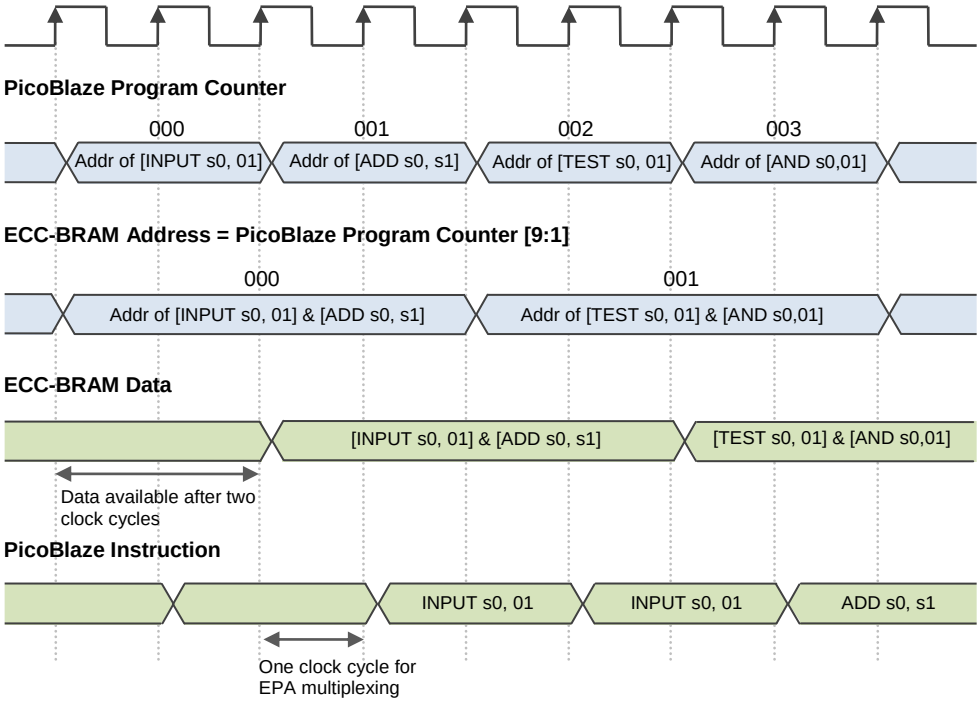


Figure 0.1 ECC-PicoBlaze non-branching operations

JUMP operation

The operation is branched when the flow control instructions are executed, which can cause instructions out-of-order. To solve this problem, corresponding solutions are presented to deal with each flow control instruction, including JUMP, CALL, RETURN and INTERRUPT. In standard PicoBlaze operation, the JUMP operation changes the PC to point towards the desired instruction address in the next clock cycle (see Figure 5.2). However if the 3 clock cycle delay exists, there will be a 2-cycles blank period (pipeline bubble) before the instruction at the new address becomes available. In such a case, a NOP instruction is manually inserted to fill this blank. The assembler instruction “LOAD s0, s0” is used as the NOP instruction, which re-assigns the register s0 to itself, having no effect on any registers or flags or the PC. Figure 0.2 gives an example of the ECC-PicoBlaze JUMP operation. After “JUMP 1F0” is executed, the PC changes to 1F0 in the next cycle. While the PicoBlaze is waiting for the new instruction at 1F0 to become available, the NOP instruction is inserted to fill the blank period. As a result, although the required

timing is met, the NOP instruction induces a 2-cycles time overhead each time a branching instruction is executed.

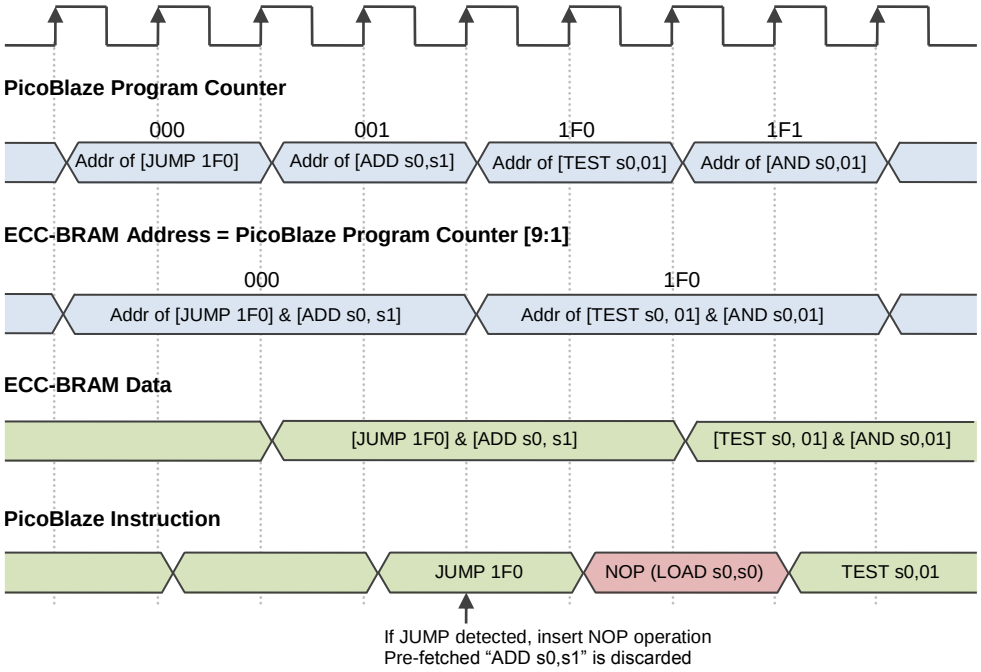


Figure 0.2 ECC-PicoBlaze JUMP operation

It should be noted that some conditional JUMP operations only change the PC when the condition is met. The conditional JUMP instructions include JUMP C (JUMP only if Carry Flag is one), JUMP NC (JUMP only if Carry Flag is zero), JUMP Z (JUMP only if Zero Flag is one), and JUMP NZ (JUMP only if Zero Flag is zero). To detect whether or not the condition is met, the two flag signals, i.e. zero flag and carry flag, are extracted from the PicoBlaze core and used by the EPA to decide whether or not a NOP insertion is needed.

CALL and RETURN operation

The CALL instruction is used to call a subprogram and the RETURN operation is used at the end of the subprogram to return back to the previous address when the program was called. In the standard operation, if a CALL instruction is executed, the

subsequent program address (current PC + 1) is pushed into the stack; meanwhile the PC is loaded with the new program address, which is the starting address of the subprogram. Similar to the JUMP instruction, since the program address is changed, a NOP operation is inserted in the same way. When the subprogram finishes, the RETURN instruction is executed to pop up the pushed PC address from the stack. Figure 0.3 gives a comparison between standard and ECC PicoBlaze timing when the CALL instruction is executed. The standard PicoBlaze pushes the subsequent address (001) into the stack and pops it back when a RETURN is executed. However, due to the 3-cycles delay, the PC pushed by the ECC-PicoBlaze is incremented by 1 (from 001 to 002), which is undesirable and results in an incorrect returned address. Therefore, when a RETURN operation is detected, the EPA is capable of decrementing the popped address by 1, whereby the returning address can be manually amended (see Figure 0.4). Besides this, another NOP operation is inserted when the program returns, since the PC is branched. Similar to conditional JUMP operations, conditional CALL operations include CALL C (CALL only if Carry Flag is one), CALL NC (CALL only if Carry Flag is zero), CALL Z (CALL only if Zero Flag is one), and CALL NZ (CALL only if Zero Flag is zero). To detect the conditions, two flag signals are used in the same way as in the conditional CALL operations.

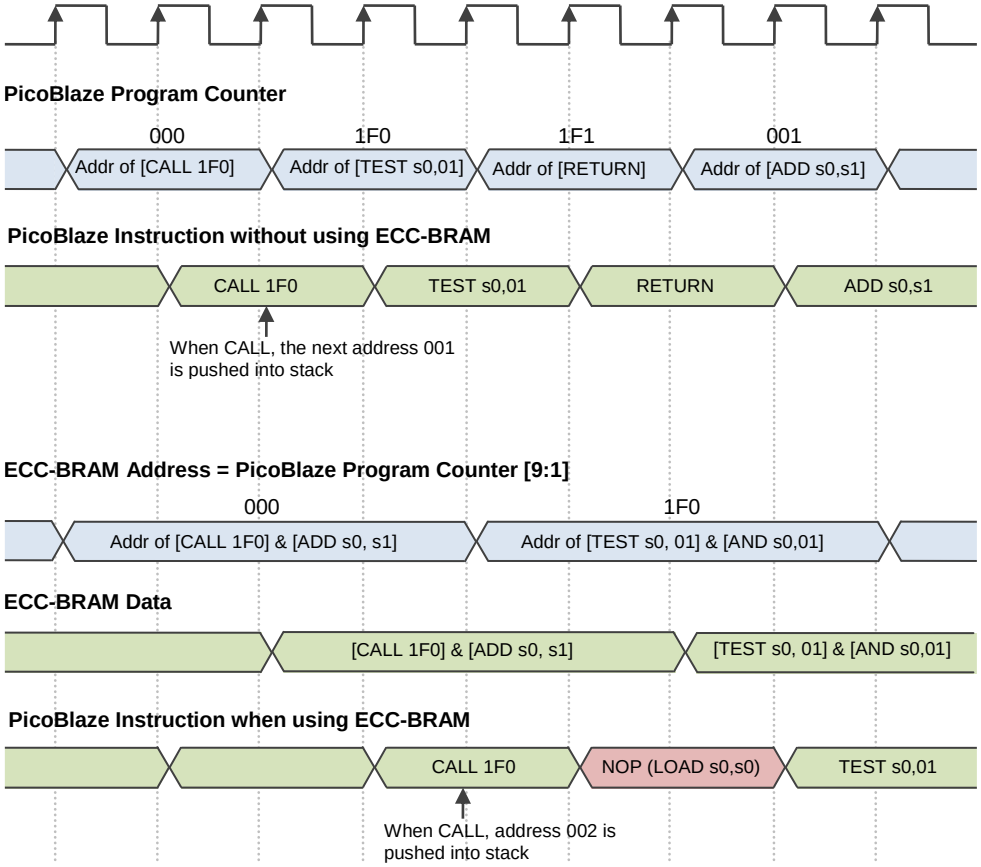


Figure 0.3 ECC-PicoBlaze CALL operation

INTERRUPT and RETURNI operation

When the PicoBlaze receives an interrupt signal, it directly changes its PC to 3FF, where a JUMP instruction can be used to change the PC to the start address of the interrupt service routine (ISR). For example, in Figure 0.5, when the PicoBlaze detects an interrupt, it will preempt the next instruction at 008, and replace it with an NOP operation before the instruction at 3FF becomes available. The preempted instruction at 008 will be re-executed as soon as the program returns from the ISR using the RETURNI instruction (see Figure 0.6). However, when using the ECC-BRAM, not only does the instruction at 008 need to be preempted, but also its previous instruction at 007 has to be stored by the EPA, or otherwise the pre-fetched instruction at 007 is lost.

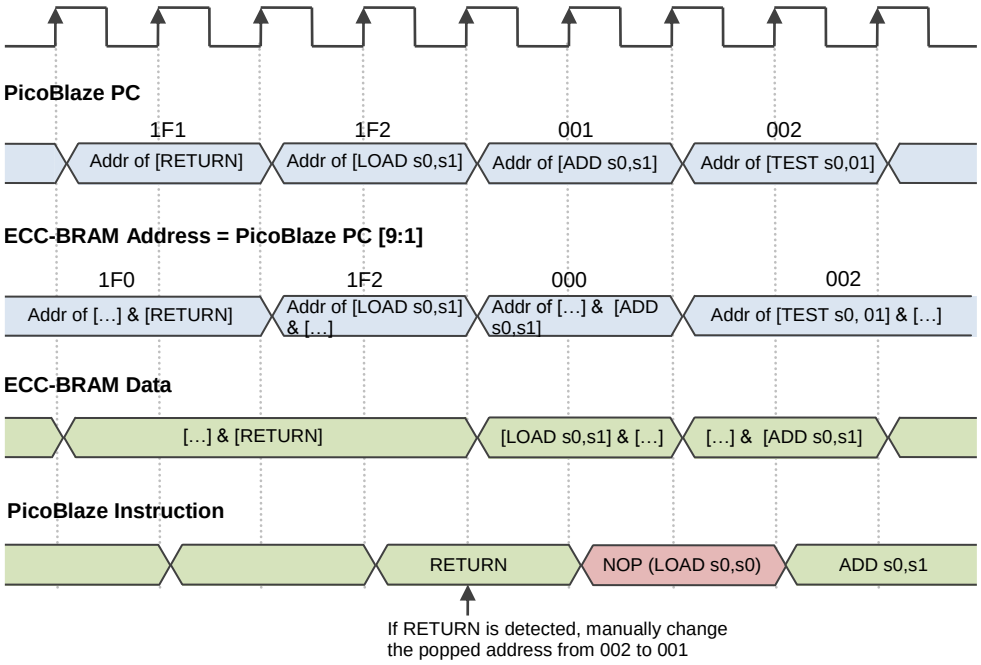


Figure 0.4 ECC-PicoBlaze RETURN operation

However, the return from an ISR, i.e. RETURNI, is different from the regular CALL & RETURN. In the previous RETURN from a called subprogram, the previously restored PC is popped from the stack, and then decremented by the EAP, so that the PC = previous PC-1. However, when returning from an interrupt, the instruction needing to be restored is not necessarily the instruction at the decremented PC. This means that the preempted instruction is not the desired one. To counter this, the instruction before the ISR starts is firstly preempted (see Figure 0.5), and then the preempted instruction is resumed when RETURN is executed, rather than returning to a previous address. For example, in Figure 0.6, when RETURNI is executed, the previous stacked PC 008 (preempted by the PicoBlaze) is popped back. Before the data in ECC-BRAM becomes available, the instruction INPUT s0, 01 (preempted by the EPA) is inserted.

Last, but not least, not only do branching operations need to be modified, but also the reset timing needs to be adjusted. In the standard PicoBlaze, the instruction at address 0 is fetched as soon as the reset signal is released, whereas in the ECC-

PicoBlaze the reset signal has to be delayed internally for another cycle so that it can synchronise with the ECC-BRAM.

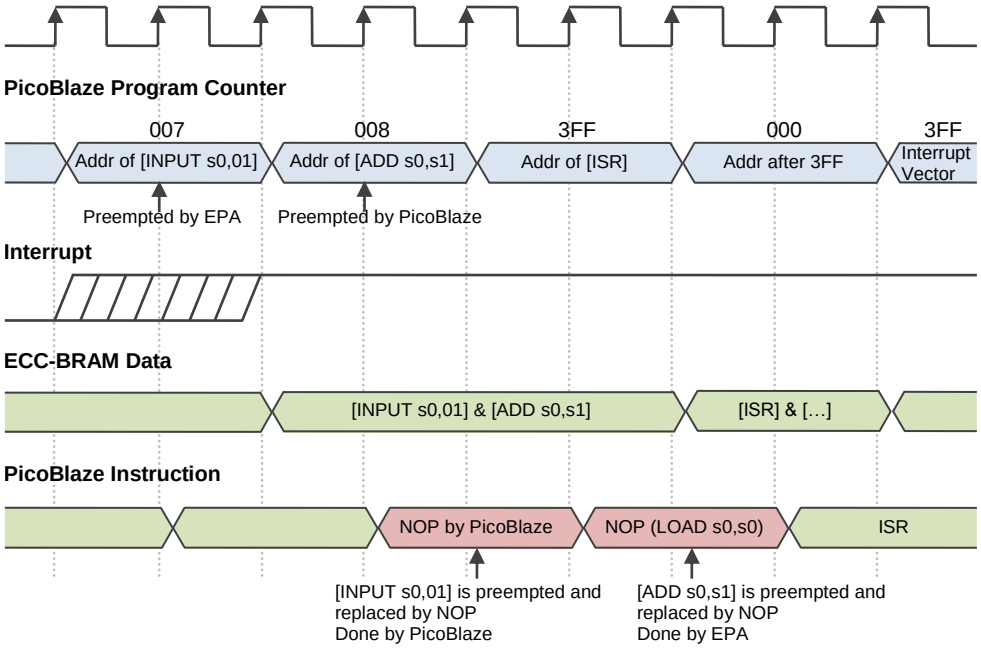


Figure 0.5 ECC-PicoBlaze INTERRUPT operation

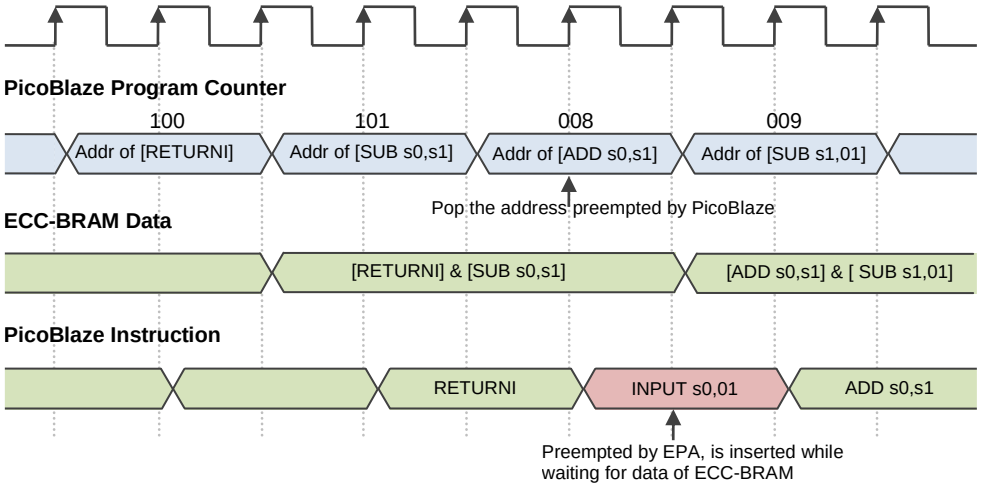


Figure 0.6 ECC-PicoBlaze RETURNI operation

Appendix-2

Development Hardware and Software Tools

The programing languages using in this project, as well as hardware platform, e.g. boards, desktops, and software CAD packages are listed below

Simulation of algorithm (section 4.1.3 and 4.2.4):

- Language: Programed in standard ANSI C
- Compiler: GCC compiler
- Programing IDE: Eclipse IDE (Helios Packages)
- Test CPU: Intel Core Due CPU at 2.4GHz

Implementation of algorithms (section 5.2)

- Hardware language: VHDL
- Hardware synthesis tool: Xilinx XST (in Xilinx ISE 13.2 package)
- Software language: PicoBlaze Aseembler [Xilinx2011b]
- Test chip: Xilinx Virtex4 FX60 FPGA [Xilinx2008]
- Test board: Xilinx ML403 [Xilinx2008]

System demonstration a): K-Nearest Neighbour Classifier (K-NN) (section 6.1)

- Hardware language: VHDL
- Hardware synthesis tool: Xilinx XST (in Xilinx ISE 13.2 package)
- Software language: PicoBlaze Aseembler [Xilinx2011b]

C (for Xilinx Microblaze microprocessor)

- Test chip: Xilinx Virtex4 FX60 FPGA [Xilinx2008]
- Test board: Xilinx ML403 [Xilinx2008]

System demonstration b): Sequence Alignment (SA) (section 6.2)

- Hardware language: VHDL
- Hardware synthesis tool: Xilinx XST (in Xilinx ISE 13.2 package)
- Software language: PicoBlaze Aseembler [Xilinx2011b]

  C (for Xilinx Microblaze microprocessor)
- Test chip: Xilinx Virtex5 LX110T FPGA [Xilinx2012a]

  Note: Virtex5 is not used for K-NN demo, as hardware was not available at that time.
- Test board: Alpha Data ADM-XRC-5LX [Xilinx2012a]